

Chapter 8: Files and Security

All programs and applets we created up to this point had one feature in common: as soon as the program or applet finishes, all information created or manipulated is lost. That is, of course, because we did not discuss reading or writing data to and from files. Therefore, we will now focus on file input and output, i.e. on writing data to file and on retrieving data from file. This will work particularly well for stand-alone programs, while applets will encounter security restrictions imposed by the JVM that will prevent it from writing data to disk (reading will work, albeit in a restricted fashion). We will tackle solutions that can overcome these applet security restrictions in the final chapter of the book.

Section 9.1 will give a brief overview of the classes relating to input/output. Section 9.2 will introduce byte-level data I/O, while section 9.3 will discuss character-level I/O. Section 9.4 will show how to save and retrieve entire objects, which is very handy and perhaps the preferred way to store and retrieve data using Java. Section 9.5 will discuss the applet sandbox and the problems associated with reading and writing data from applets. Section 9.6 will start an extended example whose conclusion can be found in chapter 10. This section is optional. Section 9.7 will explain some system-level I/O streams and explain in detail how the `Console` class works that was introduced in section 2.4, example 5. This section is also optional. Finally, section 9.7 will briefly comment on Random Access Files but will not contain any details about this particular file type.

When trying to store data in a file all programming languages face a common challenge. Data stored inside a running program is always tied to a data type. That is, we can not just store, say, the value 3.1415 in a running program, we must attach that value to a variable of a specific type (in this case a `double`, of course). That is convenient, because it will always be clear what type the data has: the type of the variable it is stored in. On the other hand, when storing data in a file, we *only* want to store the data, not the particular variable that contains the data. In other words, if a running program contains a `double` variable `x` with value 3.1415, we want to store only the value 3.1415 in the file, not the variable `x`. Later, when reading the data back from the file, we may choose to retrieve the value 3.1415 into a variable named `pi`, or even store it inside a field of some class. Therefore, our problem is that when we want to retrieve data from a file, we must also somehow know the type that the data has in order to retrieve it correctly.

For example, if someone gives you a diskette containing a file named "cool_stuff", then this file is pretty much useless to you. You need to know *something* about the content of the data file before you can correctly read it. If you knew, for example, that the file was created with Microsoft Word, you could open it with that program. If you knew that the file contained 365 `double` values indicating the average temperature for each day of the year 1999 you could then process the data correctly by writing a suitable program. Without knowing at least something, there is very little you can do with data stored in a file.

There are multiple ways of communicating to a user or program what the type of data is that is stored in a particular file. A common method for Windows and Unix systems is to use the extension of the file name, i.e. the letters after the last period, to indicate the file type. For example, a file that ends in the name ".doc" is usually a Microsoft Word document, a file ending in ".ps" is usually a file containing commands in the PostScript language, and files ending in ".txt" are usually plain text files. Macintosh computers use a slightly different approach by storing a particular four-letter code in a special part of a file that indicates the program that was used to create the file. That code is not part of the file name, but part of the data itself stored in the program. Another approach

programmers often take is to start a data file with a "file header" containing some "meta" information about the rest of the data. For example, Microsoft Word stores a particular sequence of bytes as the first few bytes in a file. Thus, if Word is used to open a document, it can immediately recognize whether the document was created by Word or by another program.

Whatever technique is used, the effect is similar: something either in the file name or in the data itself indicates the type of the file or some details about the data in the file. Our approach will be slightly different: we are, after all, not using commercial packages to create and retrieve data files but instead our own programs. Therefore, we need to be very clear about the type and format of the data we are saving in order to retrieve it successfully. We will carefully describe the data to be stored in a file in words before proceeding with creating or retrieving data.

In fact, for "industrial strength" programs we should create a separate document in a common word processing or text format describing all data file formats in detail. That document should be saved together with the data file to ensure we can always determine the type of data to be stored. Of course, the preferred format for such documentation would be an `HTML` document and we can use the facilities of `javadoc` to help create such a file quickly.

Quick View

Here is a quick overview of the topics covered in this chapter.

9.1. Overview of `java.io`

Data Streams; Basic Data Type I/O: Byte-Level Input/Output Classes; Character and String I/O: Character-Level Input/Output Classes; Object I/O: Object-Level Input/Output Classes

9.2. Saving and Retrieving Byte-Level Data

Saving Byte-Level Data; Retrieving Byte-Level Data; Using Buffered Streams for Improved Efficiency; Handling a File

9.3. Saving and Retrieving Character Data

Retrieving Character Data; Saving Character Data; A Simple Text Editor Program

9.4. Saving and Retrieving Object Data

Saving Object Data; Retrieving Object Data; The `Serial` in `Serializable`: The "Jane meets Jimmy" Example

9.5. Security, URL's, and Applets

Security Restrictions for Applets; The Applet Sandbox; Reading Data from Applets; URLs and the `URL` Class; Bridges between Character and Byte Streams

Case Study: The Online Ordering Example

(*) 9.7. System I/O Streams

(*) 9.8. `RandomAccessFile`

(*) These sections are optional

8.1. Overview of `java.io`

Java has a wealth of classes dedicated to data input and output, combined for the most part into the `java.io` package. In fact, that package contains so many classes, interfaces, and exceptions (over 60) that it can be quite intimidating trying to pick just the right one to solve a particular task. However, not all I/O classes are equally useful, and those that are most useful can be grouped into three categories, which will make it much easier trying to decide which classes to use.

This section will provide a brief overview of the most frequently used I/O classes and explain how they can be categorized for easier reference. In subsequent sections we will provide the details and examples necessary to use the classes introduced here.

Data Streams

Java uses the concept of streams to transfer data, and the `java.io` package provides class to handle input and output through such streams. This concept makes these classes sufficiently flexible to handle saving and retrieving data to and from "regular" files, as well as data transfer through network connections.

Definition 9.1.1: Data Stream

A data stream is a sequential conglomeration of bytes. Streams can be attached to data sources or data sinks on one end and to classes from the `java.io` package on the other end.

- Streams attached to data sources are used to read data from the source. Java I/O classes attached to such streams can retrieve bytes from the stream and the stream replenishes the bytes from its data source, if possible. These streams are called *input streams*.
- Streams attached to data sinks are used to write data to the sink. Java I/O classes attached to such stream insert bytes into the stream and the stream sends them to the data sink, if possible. These streams are called *output streams*.

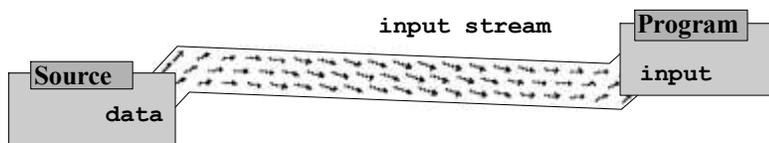


Figure 9.1.1: Input Stream connecting data source to I/O class for input

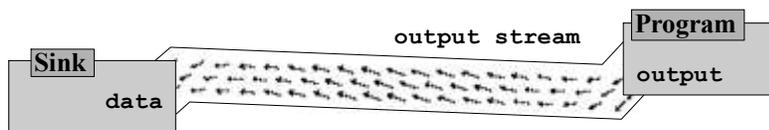


Figure 9.1.2: Output Stream connecting I/O class to data sink for output

Streams are the basis for almost all classes in the `java.io` package. That package, in fact, has a multitude of classes and it can be confusing trying to determine which class does what. We will first group the most useful of the classes into several categories, then explain the classes within each category in more detail. Not all classes from the `java.io` package are explained here and you should refer to the Java API for additional details.

Figures 9.1.3, 9.1.4, and 9.1.5 show a brief "grouped" overview of the Java streaming classes we will be covering in this chapter:

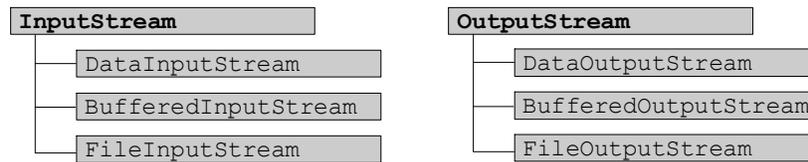


Figure 9.1.3: Input and Output Stream classes (byte-level I/O classes)¹

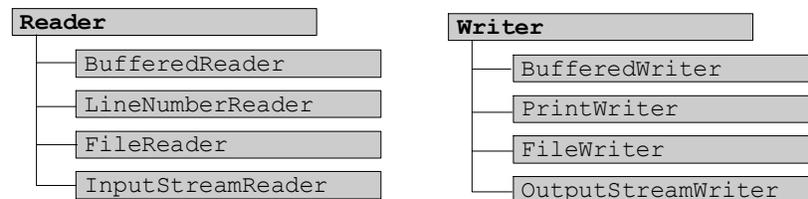


Figure 9.1.4: Reader and Writer stream classes (character-level I/O classes)²



Figure 9.1.5: Object Input and Output stream classes (object-level I/O classes)³

Basic Data Type I/O: Byte-Level Input/Output Classes

Java provides several classes to transfer bytes. These classes include a variety of methods to save and retrieve bytes and basic Java data types in a Java-native encoding. That means that data written through byte-level output streams can be read by byte-level input streams. Data written by other applications such as a plain text editor may not be compatible with these data streams. The classes should be considered in pairs, i.e. for every output class to write data there is a corresponding class to read data and visa versa.

Here are the basic classes to write byte-level data:

Class Name	Basic Description
<code>OutputStream</code>	Abstract superclass class representing an output stream of bytes.
<code>BufferedOutputStream</code>	Adds the ability to buffer the output to an output stream. Data is written into an internal buffer, and then written to the underlying stream if the buffer reaches its capacity, the stream is closed, or the buffer output stream is explicitly flushed.
<code>DataOutputStream</code>	Lets an application write primitive Java data types to an output

¹ Byte-level classes are described in detail in section 9.2.

² Character-level classes are described in detail in section 9.3.

³ Object-level classes are described in detail in section 9.4.

	stream in a portable way.
<code>FileOutputStream</code>	Output stream for writing data to a <code>File</code> or to a <code>FileDescriptor</code> .

Table 9.1.6: Classes for writing byte-level data to streams

The corresponding classes for reading data written with the above classes are:

Class Name	Description from API
<code>InputStream</code>	Abstract superclass class representing an input stream of bytes.
<code>BufferedInputStream</code>	Adds the ability to buffer the input to another input stream. As bytes from the stream are read or skipped, an internal buffer is refilled as necessary from the attached input stream.
<code>DataInputStream</code>	Lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
<code>FileInputStream</code>	Obtains input bytes from a file in a file system

Table 9.1.7: Classes for reading byte-level data from streams

Character and String I/O: Character-Level Input/Output Classes

Java contains several useful classes to read and write character-based data. That data is more uniformly standardized than byte-level data so that character-level input classes could also read and interpret data written by, say, a standard text editor. Conversely, data written by character-level output classes could be used in other programs such as standard text editors.

The classes can again be considered in pairs, i.e. for most output classes to write data there is a corresponding class to read that data.

Class Name	Description from API
<code>Writer</code>	Abstract class for writing to character streams.
<code>BufferedWriter</code>	Writes text to a character-output stream, buffering characters.
<code>PrintWriter</code>	Print formatted representations of objects to a text-output stream. Methods in this class never throw I/O exceptions but clients can check whether errors occurred using <code>checkError</code>
<code>OutputStreamWriter</code>	A bridge from character streams to byte streams. It translates characters to bytes and writes the bytes to the stream
<code>FileWriter</code>	Convenience class for writing character files.

Table 9.1.8: Classes for writing character-based data to streams

The corresponding classes for reading data written with the above classes are:

Class Name	Description from API
<code>Reader</code>	Abstract class for reading character streams.
<code>BufferedReader</code>	Reads text from a character-input stream, buffering characters as necessary.
<code>LineNumberReader</code>	A buffered character-input stream that keeps track of line numbers.
<code>InputStreamReader</code>	A bridge from byte streams to character streams: It reads bytes and translates them into characters.
<code>FileReader</code>	Convenience class for reading character files.

Table 9.1.9: Classes for reading character-based data from streams

Object I/O: Object-Level Input/Output Classes

Java, being exclusively object-oriented, also provides facilities to easily write entire objects to a stream and recover them later from a stream. This method also works through a network connection. Objects who should be transferred through streams must implement the `Serializable` interface which is part of the `java.io` package. That interface actually does not contain any methods at all, and in particular none that must be implemented if a class wants to implement `Serializable`. The purpose of implementing this interface is to actively accept serialization, i.e. if an object implements `Serializable` it agrees to be transferred through streams, if necessary.

As in the previous cases, object input/output classes also come in pairs:

Class Name	Description from API
<code>ObjectOutputStream</code>	Writes Java objects and basic data types to an <code>OutputStream</code> . Only objects that support the <code>java.io.Serializable</code> interface can be written to streams. The method <code>writeObject</code> is used to write an object to the stream. The objects must be read back from the corresponding <code>ObjectInputStream</code> with the same types and in the same order as they were written.

Table 9.1.10: Class for writing object data to streams

Class Name	Description from API
<code>ObjectInputStream</code>	Reads objects previously written using an <code>ObjectOutputStream</code> . Only objects that support the <code>java.io.Serializable</code> interface can be read from streams. The method <code>readObject</code> is used to read an object from the stream. Java's type casting should be used to get the desired type of the original object back.

Table 9.1.11: Class for reading objects from streams

The above streams are either for input (reading from) or for output (writing to). There is another class in `java.io` that represents a stream for simultaneous input and output: the `RandomAccessFile` class. We will only briefly mention that class in a later section. For now, we will focus on the details of the classes mentioned so far.

8.2. Saving and Retrieving Byte-Level Data

We will first discuss how to save and retrieve the basic data types that Java offers.

Saving Byte-Level Data

Example 9.2.1:

Create a simple program to store a double, an int, a boolean, a char, and a String value to a file named "sample.dat", in this order.

Before we can create this program, we need to find out (at least) two things:

- which methods should we use to write these basic data types to a stream
- how do we attach a stream to a file for writing to the file

According to table 9.1.6 the `DataOutputStream` might have the appropriate methods, so here is the definition of that class:

Definition 9.2.2: The `DataOutputStream` Class

The `DataOutputStream` lets an application write primitive Java data types to an output stream in a portable way⁴. They can be read back by using a `DataInputStream`. The Java API defines `DataOutputStream` as follows:

```
public class DataOutputStream extends FilterOutputStream
    implements DataOutput
{ // constructor
  public DataOutputStream(OutputStream out)
  // selected methods
  public final void writeBoolean(boolean v) throws IOException
  public final void writeByte(int v) throws IOException
  public final void writeChar(int v) throws IOException
  public final void writeInt(int v) throws IOException
  public final void writeDouble(double v) throws IOException
  public final void writeBytes(String s) throws IOException
  public final void writeUTF(String str) throws IOException
}
```

Note: The method `writeUTF` writes a String using UTF-8 encoding which results in a machine-independent representation of the String.

These methods certainly seem to be appropriate but before we can use this class we need to create an `OutputStream` to be used as input to the constructor of a `DataOutputStream`. Definition 9.2.2 also does not mention anything about attaching the output stream to a file. Thus, consulting table 9.1.6 again we will next define the class `FileOutputStream`:

Definition 9.2.3: The `FileOutputStream` Class

A `FileOutputStream` is used to attach an output stream to a file for writing data. The Java API defines `FileOutputStream` as follows:

```
public class FileOutputStream extends OutputStream
{ // selected constructors
  public FileOutputStream(String name)
    throws FileNotFoundException
  public FileOutputStream(String name, boolean append)
    throws FileNotFoundException
  public FileOutputStream(File file) throws IOException
}
```

⁴ The data is portable which means that it is written in a format that is independent of the particular operating system.

WARNING: If you create a `FileOutputStream` using an existing file name, the existing file is erased without warning, and replaced by a new, empty file. You can use methods provided by the `File` or `FileDialog` class⁵ to ensure that you do not accidentally erase an important file.

Now we have the ingredients to proceed. We can use a `FileOutputStream` to create an instance of an output stream that is attached to a file for writing and we can use that as input to a `DataOutputStream` so that we have access to the methods necessary to write the respective data.

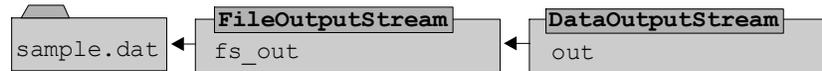


Figure 9.2.1: Instantiating a `DataOutputStream` on a `FileOutputStream`

As constructor for the `FileOutputStream` we select the one requiring a single string representing the file name. Here is the code:

```

import java.io.*;

public class SimpleOutputTest
{
    public static void main(String args[])
    {
        double Pi = 3.1415;
        int i = 10;
        boolean okay = true;
        char cc = 'J';
        String s = "Java by Definition";
        try
        {
            FileOutputStream fs_out = new FileOutputStream("sample.dat");
            DataOutputStream out = new DataOutputStream(fs_out);
            out.writeDouble(Pi);
            out.writeInt(i);
            out.writeBoolean(okay);
            out.writeChar(cc);
            out.writeUTF(s);
            out.close();
        }
        catch(FileNotFoundException fe)
        {
            System.err.println(fe);
        }
        catch(IOException ioe)
        {
            System.out.println(ioe);
        }
    }
}
  
```

When the program executes, no output will be displayed (unless some error occurs during file creation). Instead, a file named "sample.dat" will be created, containing the corresponding data. ■

Retrieving Byte-Level Data

If we tried to open this data file with another program such as a plain text editor, we would not be able to easily understand the data. It is written in a format that will be understood if the file is read using a `DataInputStream`, as in example 9.2.4.

Example 9.2.4:

⁵ See definition 9.2.12 and definition 9.3.5.

Create a simple program to read a double, an int, a boolean, a char, and a String value from a file named "sample.dat". The data in that file must have been created using a `DataOutputStream`, such as in example 9.2.1.

We already know how to attach an output stream to a file for writing. Now we need to investigate the classes to create an input stream to a file for reading and we need to find the corresponding methods to read the various data types. As far as the methods are concerned, they are part of the complementary method to `DataOutputStream`:

Definition 9.2.5: The `DataInputStream` Class

This class lets an application read primitive Java data types from an underlying input stream in a machine-independent way. The data should have been written by a `DataOutputStream`. The Java API defines `DataInputStream` as follows:

```
public class DataInputStream extends FilterInputStream
    implements DataInput
{
    // constructor
    public DataInputStream(InputStream in)
    // selected methods
    public final int skipBytes(int n) throws IOException
    public final boolean readBoolean() throws EOFException, IOException
    public final byte readByte() throws EOFException, IOException
    public final char readChar() throws EOFException, IOException
    public final int readInt() throws EOFException, IOException
    public final double readDouble() throws EOFException, IOException
    public final String readUTF() throws EOFException, IOException
}
```

If any of these methods attempt to read past the end of the stream, an `EOFException` will be generated.

Similar to the `DataOutputStream`, the constructor of a `DataInputStream` needs an `InputStream` attached to a file before we can use these methods. The `FileInputStream` will provide that stream:

Definition 9.2.6: The `FileInputStream` Class

This class obtains input bytes from an existing file and returns an `InputStream` attached to that file. The Java API defines `FileInputStream` as follows:

```
public class FileInputStream extends InputStream
{
    // selected constructors
    public FileInputStream(String name) throws FileNotFoundException
    public FileInputStream(File file) throws FileNotFoundException
}
```

Now we can read the data file "sample.dat" that was created in example 9.2.1 as follows:



Figure 9.2.2: Instantiating a `DataInputStream` on a `FileInputStream`

```

import java.io.*;

public class SimpleInputTest
{
    public static void main(String args[])
    {
        try
        {
            FileInputStream fs_in = new FileInputStream("sample.dat");
            DataInputStream in = new DataInputStream(fs_in);
            double Pi = in.readDouble();
            int i = in.readInt();
            boolean okay = in.readBoolean();
            char cc = in.readChar();
            String s = in.readUTF();
            in.close();
            System.out.println("Pi = " + Pi + ", i = " + i);
            System.out.println("okay = " + okay + ", cc = " + cc);
            System.out.println("s = " + s);
        }
        catch(FileNotFoundException fnfe)
        {
            System.err.println(fnfe);
        }
        catch(IOException ioe)
        {
            System.err.println(ioe);
        }
    }
}

```

When this program executes, it reads the original values from the data file and writes them to standard output. Since we used a `DataOutputStream` to create the file, we know exactly the types we want to read at what time. No exception should be generated and the program should terminate normally.⁶ ■

The data file created by a `DataOutputStream` is system-independent and portable. That means that if the data was written by a `DataOutputStream` on one platform, say under Windows, it can be copied to another system such as a Macintosh computer and a Java program using a `DataInputStream` on that platform will be able to restore the data correctly. The file must be transferred as a binary file from one platform to the other.

We will later see how to create a complete GUI-based program that uses standard "File | Open" and "File | Save" dialog boxes to select the file names. For now, we will further experiment with simple programs to explore additional details about dealing with streams.

Example 9.2.7:

Create a program that writes a random number of random `double` values to a file named "doubles.txt". Then create a second program to read all numbers back and find their average.

If you write a sufficiently large number of `double` random values between 0 and 1, can you guess the approximate value of the average you should be getting?

The first program, clearly, is easy and offers nothing new (except for recalling how to create random numbers):

```

import java.io.*;

```

⁶ We know that the first value in the file is of type `double` because we know that program used to create the data file. We will see in example 9.2.14 what can happen when we attempt to read a data type that is different from the one that was written, and in example 9.3.10 how a such a file might look when opened with a standard text editor.

```
public class RandomWriteTest
{   public static void main(String args[])
    {   try
        {   FileOutputStream fs_out = new FileOutputStream("sample.dat");
            DataOutputStream out = new DataOutputStream(fs_out);
            for (int i = 0; i < (int)(Math.random()*10000) + 10000; i++)
                out.writeDouble(Math.random());
            out.close();
        }
        catch(FileNotFoundException fe)
        {   System.err.println(fe); }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
}
```

The problem when trying to read the numbers back is that we don't know how many numbers the file contains. Thus, we will use the `EOFException` to inform us about the end of file, thus signifying that the last number has been read.⁷ Once that trick has been determined, the rest of the code is easy:

```
import java.io.*;

public class RandomReadTest
{   public static void main(String args[])
    {   try
        {   FileInputStream fs_in = new FileInputStream("sample.dat");
            DataInputStream in = new DataInputStream(fs_in);
            double sum = 0.0;
            int count = 0;
            try
            {   while (true)
                {   sum += in.readDouble();
                    count++;
                }
            }
            catch(EOFException eofe)
            {   System.out.println("Counted: " + count);
                System.out.println("Average: " + (sum / count));
            }
            in.close();
        }
        catch(FileNotFoundException fe)
        {   System.err.println(fe); }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
}
```

When this program executes, it will read back all numbers previously saved to file and compute their average. Since the numbers that were saved were random numbers between 0 and 1, all numbers above 0.5 have the same chance of appearing as those below 0.5. Thus, the average value should be approximately 0.5.

⁷ This is somewhat similar to example 8.1.20 where we pop numbers from a stack until a `StackException` tells us that the stack is empty.

```

CommandOutput
C:\Java\09\9.2.7>java RandomWriteTest

C:\Java\09\9.2.7>java RandomReadTest
Counted: 10085
Average: 0.499678748622025

```

Figure 9.2.3: Writing and reading random numbers between 0 and 1

Using Buffered Streams for Improved Efficiency

Our next example in this section will examine the usefulness of buffering the input and output streams. A buffered stream is a stream with an additional internal buffer. When a single write request is issued to a buffered stream, the data is not necessarily written to the attached output stream or file. Instead, the data is written to the buffer at high speed. When the buffer is full or the stream is closed, all data is written in one operation from the buffer to the output stream or file. Therefore, instead of many "small" write requests to disk, only few "large" write requests are performed, improving the efficiency of writing data to a file. Similarly, a read request to a buffered stream attached to an input stream may not only read the amount of data requested, but instead as much data as can fit into the buffer. Subsequent read requests are then satisfied from the buffer, not from an attached file, again improving the efficiency of reading data.

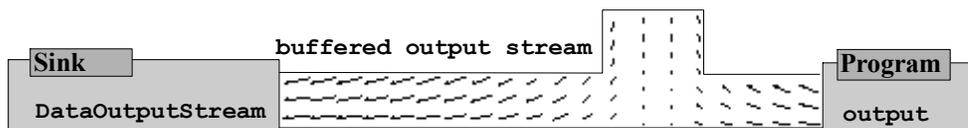
Definition 9.2.8: The `BufferedOutputStream` Class

A `BufferedOutputStream` adds the ability to buffer the output to another output stream. Data is written into an internal buffer, and only written to the underlying stream if the buffer reaches its capacity, the stream is closed, or the buffered output stream is explicitly flushed. All methods of an attached output stream will automatically be buffered.

```

public class BufferedOutputStream extends FilterOutputStream
{
    // selected constructor
    public BufferedOutputStream(OutputStream out)
    // selected method
    public void flush() throws IOException
}

```

Figure 9.2.4: Visualization of `BufferedOutputStream`

Definition 9.2.9: The `BufferedInputStream` Class

A `BufferedInputStream` adds the ability to buffer the input to another input stream. As bytes from the stream are read or skipped, an internal buffer is refilled as necessary from the attached input stream. All methods of the attached input stream will automatically be buffered.

```

public class BufferedInputStream extends FilterInputStream
{ // selected constructor
  public BufferedInputStream(InputStream in)
}

```

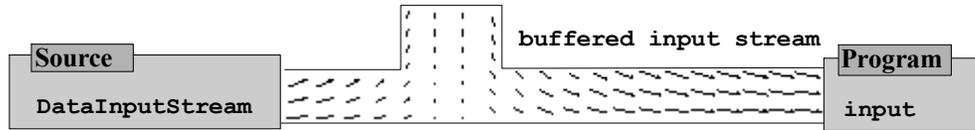


Figure 9.2.5: Visualization of BufferedInputStream

Example 9.2.10:

Create a program to write 100,000 random `double` numbers to a file. Use a buffered as well as an unbuffered output stream and measure the time difference for the operation, if any. Do the same for reading 100,000 `double` numbers.

The program to write the doubles to disk is just as before. Before starting the write operation we obtain the current time and compare it with the time at the end of the operation to determine how long everything took:



Figure 9.2.6: Instantiating a DataOutputStream on a FileOutputStream

```

import java.io.*;

public class WriteUnbufferedTest
{ public static void main(String args[])
  { try
    { long start = System.currentTimeMillis();
      FileOutputStream fs_out = new FileOutputStream("sample.dat");
      DataOutputStream out = new DataOutputStream(fs_out);
      for (int i = 0; i < 100000; i++)
        out.writeDouble(Math.random());
      out.close();
      long stop = System.currentTimeMillis();
      System.out.println("Time passed: " + (stop - start));
    }
    catch(IOException ioe)
    { System.out.println(ioe); }
  }
}

```

In a sample run on my particular system, the time it takes to write all data (which, incidentally, takes exactly 800,000 bytes) is 17,300 milliseconds, or just over 17 seconds.

To create a buffered version of this program is very simple: we just wrap the file output stream in a buffered output stream before attaching it to the data output stream.



Figure 9.2.7: Instantiating a DataOutputStream on a FileOutputStream via a BufferedOutputStream

The modified code is printed in bold and italics:

```
import java.io.*;

public class WriteBufferedTest
{ public static void main(String args[])
  { try
    { long start = System.currentTimeMillis();
      FileOutputStream fs_out = new FileOutputStream("sample.dat");
      BufferedOutputStream bfs_out = new BufferedOutputStream(fs_out);
      DataOutputStream out = new DataOutputStream(bfs_out);
      for (int i = 0; i < 100000; i++)
        out.writeDouble(Math.random());
      out.close();
      long stop = System.currentTimeMillis();
      System.out.println("Time passed: " + (stop - start));
    }
    catch(IOException ioe)
    { System.out.println(ioe); }
  }
}
```

We actually do not need the references `fs_out` and `bfs_out`. They are only instantiated to deliver a proper buffered stream to the `DataOutputStream` that is attached to the file "sample.dat". We could also use a single call such as:

```
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(
        new FileOutputStream("sample.dat")));
```

to open a properly buffered `DataOutputStream`.

In any case, when this program executes on the same system as the previous version, a sample run takes only 830 milliseconds. In other words, the unbuffered version executes in about 17 seconds, while the buffered version finishes in less than 1 second. That, indeed, is a significant improvement so it is well worth using a buffered output stream in most situations.

Next, we will read the data just created, first through an unbuffered and then through a buffered input stream. The first version is again straightforward, using a `DataInputStream`:



Figure 9.2.8: Instantiating a `DataInputStream` on a `FileInputStream`

```
import java.io.*;

public class ReadUnbufferedTest
{ public static void main(String args[])
  { double sum = 0;
    try
    { long start = System.currentTimeMillis();
      FileInputStream fs_in = new FileInputStream ("sample.dat");
      DataInputStream in = new DataInputStream(fs_in);
      for (int i = 0; i < 100000; i++)
        sum += in.readDouble();
      in.close();
      long stop = System.currentTimeMillis();
    }
  }
}
```

```

        System.out.println("Average: " + (sum / 100000));
        System.out.println("Time passed: " + (stop - start));
    }
    catch(IOException ioe)
    {   System.out.println(ioe); }
}
}

```

This version, on my system, takes approximately 15,270 milliseconds, or roughly 15 seconds⁸. Let's compare that against a buffered input stream, which is created similar to the buffered output stream:

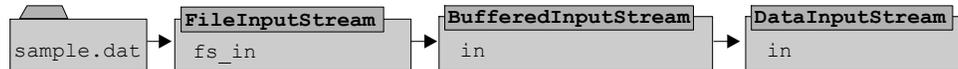


Figure 9.2.9: Instantiating a DataInputStream on a FileInputStream via a BufferedInputStream

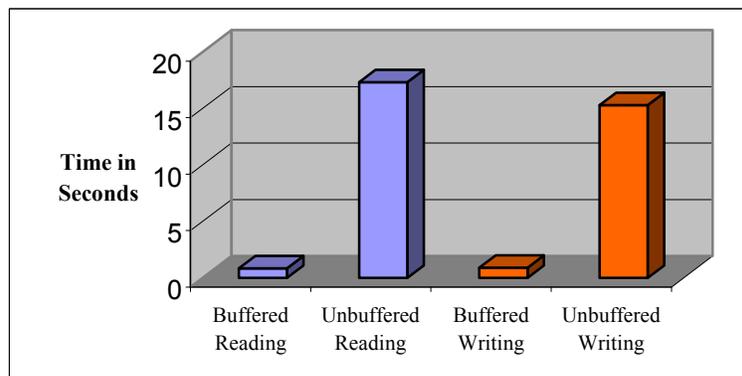
```

import java.io.*;

public class ReadBufferedTest
{   public static void main(String args[])
    {   double sum = 0;
        try
        {   long start = System.currentTimeMillis();
            FileInputStream fs_in = new FileInputStream ("sample.dat");
            BufferedInputStream bfs_in = new BufferedInputStream(fs_in);
            DataInputStream in = new DataInputStream(bfs_in);
            for (int i = 0; i < 100000; i++)
                sum += in.readDouble();
            in.close();
            long stop = System.currentTimeMillis();
            System.out.println("Average: " + (sum / 100000));
            System.out.println("Time passed: " + (stop - start));
        }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
}

```

When this buffered version executes it finishes the entire reading operation- on the same system as the previous unbuffered version - in just under 1 second. Again, that is a tremendous performance improvement and the work to achieve that is so negligible that it is almost always worth using a buffered stream for input.



⁸ Of course, the computed average of these random numbers is again very close to 0.5, as expected.

Figure 9.2.10: Buffered versus unbuffered I/O operations

Handling a File

We now know how to transfer data using streams in and out of files. But we do not know any details about working with files such as determining the file size, the date it was created, in which directory a file is located, and so on. There are also important operations we may need to perform with a file as a whole such as renaming, deleting, or copying it. And the most obvious piece of information we may need to determine is whether a file already exists in order to prevent an existing file from being accidentally overwritten.

Example 9.2.11:

Create a program `SafeCopy` that creates an identical copy of a file. The program should accept the names of one source file and one target file on the command line. It should only copy the source file to the target file if the target file does not already exist to prevent a file from being overwritten.

In principle this program is easy: create both a `DataInputStream` and a `DataOutputStream` and copy every byte from the source file to the target file. However, we must make sure that the source file actually exists before copying from it and we need to check whether the target file does not already exist to prevent existing files from being overwritten. But the classes defined so far do not contain any methods for checking for a file's existence, so we need to introduce another class that the `java.io` package makes available for the purpose of dealing with files.

Definition 9.2.12: The `File` Class

The `File` class is used to represent a file, a directory name, or a combination of directory names and file. The file names used are highly system-dependent, but the `File` class provides convenient methods to access and manipulate files in a system-independent manner. The Java API defines `File` as follows:

```
public class File extends Object implements Serializable, Comparable
{ // constructor
  public File(String pathname)
  // selected fields
  public static final String separator
  // selected methods
  public String getName()
  public String getPath()
  public URL toURL() throws MalformedURLException
  public boolean exists()
  public boolean isDirectory()
  public boolean isFile()
  public long length()
  public boolean delete()
  public File[] listFiles()
  public File[] listFiles(FileFilter filter)
  public boolean mkdir()
  public boolean mkdirs()
  public boolean renameTo(File dest)
```

```
}

```

This class actually hides a lot of system dependent complications. For example, on Windows systems directory and file names are separated by the "\" character, on Unix system that separator is the "/" character, and Macintosh computers use ":". Whatever the case may be, inside a Java program you can refer to the separator as `File.separator` and it will always be the correct one for the underlying operating system of the current JVM.⁹ There are many additional methods available in the `File` class and much could be said as to the system-dependency of many of these methods. We refer you to the Java API for additional details.

In our current situation this class provides the `exists` method we need to check on the existence of a file before creating an input or output stream. Therefore, we can now create our `SafeCopy` program as follows:

```
import java.io.*;

public class SafeCopy
{ public static void copyFile(DataInputStream in, DataOutputStream out)
  throws IOException
  { try
    { while (true)
      { out.writeByte(in.readByte());
      }
    catch(EOFException eof)
    { return; }
  }
  public static void main(String args[])
  { if (args.length != 2)
    System.out.println("Usage: java Copy sourceFile targetFile");
    else
    { String inFileName = args[0], outFileName = args[1];
      File inFile = new File(inFileName);
      File outFile = new File(outFileName);
      if (!inFile.exists())
        System.out.println(inFileName + " does not exist.");
      else if (outFile.exists())
        System.out.println(outFileName + " already exists.");
      else
      { try
        { DataInputStream in = new DataInputStream(
          new BufferedInputStream(
            new FileInputStream(inFileName)));
          DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
              new FileOutputStream(outFileName)));

          copyFile(in, out);
          in.close();
          out.close();
        }
        catch(IOException ioe)
        { System.out.println("Unknown error: " + ioe); }
      }
    }
  }
}
```

⁹ A call to `System.getProperty("file.separator")` will also return the default separator for the underlying operating system and no import statement is necessary. Properties are not supported on older versions of Java, hence the need for the `File.separator` constant.

The program first checks for two arguments on the input line, then checks if the source file exists, and the target file does not exist. If everything is in order, buffered input and output streams are created and the actual copying is performed via the `copyFile` method. Note that the `copyFile` method uses `readByte` and `writeByte` to copy every byte of the input file to the output file. We have of course used buffered streams to improve the efficiency of the program - it would be *very* slow if we used unbuffered streams. ■

The `File` class from definition 9.2.12 can also be used to obtain information about all files in a directory. You could use it to provide a replacement for the standard DOS `dir` command, or the Unix `ls` command.

Example 9.2.13

Write a program that lists all files and directories in the directory specified on the command line to the program. Include the file size and the type "file" or "directory". Make sure the listing is sorted alphabetically, with directories appearing before files.

The `File` class is the essential class to use here. Its `isDirectory` method can determine whether a given path is a file or directory, `listFiles` will return all files and directories within a given directory, and `length` will return the file size. Armed with these methods our class is easy, not counting the sorting part:

```
import java.io.*;

public class Dir
{   private static void showDirInfo(File list[])
    {   for (int i = 0; i < list.length; i++)
        {   if (list[i].isDirectory())
            System.out.print("DIRECTORY");
            else
                System.out.print(list[i].length() + " bytes");
            System.out.println("\t" + list[i]);
        }
    }

    public static void main(String args[])
    {   File path = new File(System.getProperty("user.dir"));
        if (args.length > 0)
            path = new File(args[0]);
        if (path.exists() && path.isDirectory())
            showDirInfo(path.listFiles());
        else
            System.out.println("Path not found or not directory");
    }
}
```

The only trick is to use the `System.getProperty("user.dir")` command is to make sure that the current directory is used if the user does not provide one. But our program does not sort the file listing so far. Adding that feature, again, is simple using Java's build-in sorting mechanisms. Before the `showDirInfo` method displays its information, it should sort the array of `File` object. As explained in definitions 7.4.6 and 7.4.7, we need to create a `Comparator` that determines how two `File` objects compare, then we can use the `Arrays.sort` method to sort easily the array of `File`

objects. The `Comparator` needs to make sure that directories will be "smaller" than files. Here is the code for that class¹⁰:

```
import java.util.*;
import java.io.File;

public class FileNameSorter implements Comparator
{   public int compare(Object o1, Object o2)
    {   File f1 = (File)o1; File f2 = (File)o2;
        if (f1.isDirectory())
        {   if (f2.isDirectory())
            {   return f1.getName().compareTo(f2.getName());
                else
                {   return -1;
                    }
            }
        else
        {   if (f2.isDirectory())
            {   return 1;
                else
                {   return f1.getName().compareTo(f2.getName());
                    }
            }
        }
    }
    public boolean equals(Object o1, Object o2)
    {   return ((File)o1).getName().equals(((File)o2).getName()); }
}
```

Now we can add sorting capabilities to our directory listing class. All we have to do is add a single line as the first line of the `showDirInfo` method to ensure that the files and directories are sorted according to the criteria specified in `FileNameSorter`:

```
import java.io.*;
import java.util.*;

public class Dir
{   private static void showDirInfo(File list[])
    {   Arrays.sort(list, new FileNameSorter());
        /* rest unchanged */
    }
    public static void main(String args[])
    {   /* no changes */
    }
}
```

Figure 9.2.11 shows the files and directories of the JDK as provided by our `Dir` class:

¹⁰ Compare example 7.4.8 for sorting (and searching) objects using an appropriate `Comparator`.

```

C:\Java\09\9.2.13>java Dir \bin\prg\jdk\
DIRECTORY      C:\bin\prg\jdk\bin
DIRECTORY      C:\bin\prg\jdk\demo
DIRECTORY      C:\bin\prg\jdk\include
DIRECTORY      C:\bin\prg\jdk\include-old
DIRECTORY      C:\bin\prg\jdk\jre
DIRECTORY      C:\bin\prg\jdk\lib
945 bytes      C:\bin\prg\jdk\COPYRIGHT
8765 bytes     C:\bin\prg\jdk\LICENSE
5236 bytes     C:\bin\prg\jdk\README
320536 bytes   C:\bin\prg\jdk\Uninst.isu
19182 bytes    C:\bin\prg\jdk\readme.html

```

Figure 9.2.11: Output of directory listing provided by Dir class

Our final example of byte-level data streams will illustrate that without external knowledge of what is in a file the data contained in the file is useless.

Example 9.2.14:

Create a program that writes two integer values to a file via a data output stream, then open the same file and attempt to read the data as one double value. Does it work? Explain.

The program itself is simple, given our previous examples. The question will be whether it works:

```

import java.io.*;

public class Confused
{   public static void writeIt()
    {   try
        {   DataOutputStream out = new DataOutputStream(
            new FileOutputStream("sample.dat"));

            out.writeInt(100);
            out.writeInt(200);
            out.close();
        }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
    public static void readIt()
    {   try
        {   DataInputStream in = new DataInputStream(
            new FileInputStream("sample.dat"));

            System.out.println(in.readDouble());
        }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
    public static void main(String args[])
    {   writeIt();
        readIt();
    }
}

```

The program compiles. Here is what we get when we execute the program:

Bert G. Wachsmuth

DRAFT April 2009

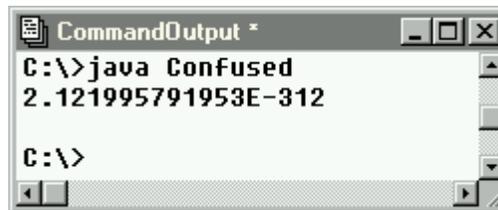


Figure 9.2.12: Output of Confused program

Surprisingly enough, no error message is generated and a double number is read from the file. For all intent and purposes, the number has a random value. However, if we did not know that the file originally represented two integer values 100 and 200, we might think that this, indeed, is the value represented by the data. And in fact that assumption is correct. There are 8 bytes contained in the data file. If interpreted as one double number, they represent the above number. If interpreted as two integers, they represent the values 100 and 200. Both interpretations are correct, and without *prior* knowledge about which interpretation is the intended one, we have *no way* of distinguishing the right from the wrong answer. ■

8.3. Saving and Retrieving Character Data

The classes discussed so far work very well to transfer values of the basic data types between Java programs, or to store data from one program for later retrieval by other programs. Frequently, however, we are interested in creating or using character-based data only. The idea is to create or use data files that are readable by "humans" as well as by a Java program. For example, we might want to do a study on global warming and we may have data values representing temperatures for the last 50 years that we obtained from an external source as a plain text file. In other words, the data file might contain double-numbers, one number per line, where each number represents a particular temperature reading. The data can be viewed with a standard text editor and might look something like this:

```
65.6
66.7
67.8
```

Data such as this can *not* be read using the `readDouble` method from a `DataInputStream`, because it does not represent double values in a format that this method will expect.¹¹

As a more useful example, suppose we want to create a text file viewer, or a simple text file editor such as `Notepad` or `SimpleText`. In that case we are concerned with creating data files that can be shared with other applications, or with using text files that have been created by other applications.

Therefore, we will now investigate the `Reader` and `Writer` classes that Java provides for just this purpose of dealing with text-based data files. Actually, the classes `Reader` and `Writer` are abstract classes and thus will be less interesting to us than their concrete subclasses `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, and `PrintWriter`.

¹¹ It can also not be read as a `String` and then converted to a double because for the `readUTF` to correctly interpret a sequence as characters as a `String` it must be written by `writeUTF`.

Retrieving Character Data

The basic input/output classes to read character data are `BufferedReader` and `PrintWriter`, working in conjunction with `FileReader` and `FileWriter` to write character data:

Definition 9.3.1: The `BufferedReader` Class

The `BufferedReader` class reads text from a character-input stream, buffering characters as necessary. The Java API defined `BufferedReader` as follows:

```
public class BufferedReader extends Reader
{ // selected constructor
  public BufferedReader(Reader in)
  // selected methods
  public String readLine() throws IOException
}
```

A line of text is considered to be terminated by any one of the characters line feed ('`\n`'), carriage return ('`\r`'), or carriage return followed immediately by linefeed. If there are no more lines contained in the stream, the `readLine` method returns null and does not through an exception.

What is considered a line of text is actually slightly different depending on the underlying operating system (which, after all, does ultimately store the data on disk). Some operating systems terminate a line with a single "line feed" character '`\n`', others use both a carriage return '`\r`' as well as a line feed character '`\n`'. In other words, if a single text file contains the lines:

```
this is the first line
this is another line
```

then the actual characters typically stored as data for the first line are:

```
this is the first line\n
```

if the text file was created on a Unix system, and

```
this is the first line\r\n
```

if the text file was created on a Windows (DOS) system. Note that this, however, is not always consistent, adding further aggravation to dealing with text files. Using Java, these differences will not matter and are dealt with automatically most of the time.

Example 9.3.2:

Use a standard text editor such as `NotePad`, `SimpleText`, or `Emacs`, to create a new text file containing these exact three lines:

```
Java
by
Definition
```

Save the file and look at the number of bytes it occupies. If you have access to different operating systems, create such a text file on every platform you have access to. Compare the number of bytes to the number of characters and explain any discrepancies.

Instead of describing how to create the file, we will assume you are familiar with that process and describe what you might find. Table 9.3.1 denotes the file sizes of a file where we typed exactly the characters above, pressing `ENTER` exactly twice, once after the word "Java" and again after "by" (but not after "Definition").

Platform	File created by	Size in bytes
Windows	Notepad	20
Windows	PFE	22
Unix	Emacs	18
Macintosh	SimpleText	380

Table 9.3.1: File sizes for different operating systems

The actual number of characters is 16, so the file will clearly be 16 bytes or more. Using `Emacs` on a Unix systems, two additional linefeed characters `'\n'` are appended when we pressed the `ENTER` key, resulting in 18 bytes total. Using `Notepad` on Windows, two pairs of carriage return plus linefeed `'\r\n'` are inserted when pressing `ENTER` twice, resulting in 20 bytes total. `PFE` (Programmer's File Editor) also seems to add two additional characters somewhere, resulting in 22 bytes total. These additional characters could be, for example, an additional `'\r\n'` pair at the end of the last line, depending on the exact configuration of `PFE`. `SimpleText` on a Macintosh adds additional data to a special section of the file, indicating the program used to create the data file as well as additional information, explaining the resulting 'large' file size. ■

Before we can start using a `BufferedReader`, we need to know how to create the proper input type to the stream:

Definition 9.3.3: The `FileReader` Class

The `FileReader` class is a convenience class to attach files to other classes requiring a `Reader` class as input. The Java API defines `FileReader` as follows:

```
public class FileReader extends InputStreamReader
{
    // selected constructors
    public FileReader(String fileName) throws FileNotFoundException
    public FileReader(File file) throws FileNotFoundException
}
```

The `FileReader` class forms the connection between the `BufferedReader` stream and the actual data file acting as source, much like the relation between `DataInputStream` and `FileInputStream`:



Figure 9.3.2: Instantiating a `BufferedReader` on a `FileReader`

Now we are ready to create a "true" program, i.e. a program that works similarly to other commercial programs you may have used before. We could try to improve on that program so that it becomes a

competitor to other existing programs with similar capabilities. And because programs written in Java execute without change on different platforms, our program has an immediate competitive advantage to programs written in, say, C++ or Visual Basic.

Example 9.3.4:

Create a simple program that can view text files in a scrollable viewing area. The program should extend `Frame` and contain a menu with a single `File` menu choice. The menu items for that choice should be `Open` and `Exit`. Choosing `Open` should bring up a standard "file open" dialog box to pick the text file to be opened. Choosing `Exit` should close the program. You may assume that the text file fits into memory.¹²

There are really three parts to this example that are new:

- How to open a plain text file for reading
- How to read a text file if we do not know how many lines of text it contains
- How to create a "standard file open" dialog

Before putting together the complete program, let's treat these pieces separately:

Opening a plain text file for reading is simple, given our above classes: the `FileReader` will provide a stream of type `Reader` connected to a file and the `BufferedReader` class will provide the method `readLine` to read lines of text. In other words, assuming that the string `fileName` contains the name of the file to open, the line:

```
BufferedReader reader = new BufferedReader(new FileReader(fileName));
```

will create a proper input stream for reading characters.

To read a text file line by line until the end of the file has been reached, we can use the fact that the `readLine` method of a `BufferedReader` will return `null` when the end of the file has been reached. Assuming `reader` is of type `BufferedReader` and has been attached to the file as above, the following code will read all available lines:

```
String line;
try
{ while ( (line = reader.readLine()) != null)
    // do something with the line read
}
catch(IOException ioe)
{ System.out.println(ioe); }
```

Finally, to create a "standard file open" dialog we should consult the `java.awt` package. After all, that package is responsible for all graphical user interface elements, so we should expect to find an appropriate class there.¹³ Indeed, that package contains the `FileDialog` class, defined as follows:

¹² Since we will put the text into a `TextArea`, the largest size possible will be about 32KB. If we wrote the program using only Swing components there would be no such limit.

¹³ If your program uses Swing components instead of the AWT you need to use `javax.swing.JFileChooser` instead of `java.awt.FileDialog`.

Definition 9.3.5: The `FileDialog` Class

The `FileDialog` class represents either a standard "File | Open" or "File | Save" dialog box from which the user can select a file for reading or writing. The dialog is modal, so once made visible via `setVisible` or `show` it will block until the user chooses a file or cancels the dialog.¹⁴ The Java API defines `FileDialog` as follows:

```
public class FileDialog extends Dialog
{ // selected constructors
  public FileDialog(Frame parent, String title)
  public FileDialog(Frame parent, String title, int mode)
  // fields
  public static final int LOAD
  public static final int SAVE
  // selected methods
  public String getDirectory()
  public void setDirectory(String dir)
  public String getFile()
  public void setFile(String file)
  public FilenameFilter getFilenameFilter()
  public void setFilenameFilter(FilenameFilter filter)
}
```

The dialog will contain a Cancel and OK button. If the user chooses Cancel, the method `getFile` will return `null`. If the dialog was created in `SAVE` mode, the user must confirm overwriting an existing name before the dialog closes.

In other words, if a `FileDialog` is instantiated using the `LOAD` constant, it will block until a user either cancels the dialog or selects a file name and clicks on `Open`. An application can use `getFile` to check for `null` to determine whether the user canceled the dialog, and it can use the `exists` method of the `File` class to check if the filename exists. Figure 9.3.3 shows a standard "File | Open" dialog box for the Windows 98 operating system:

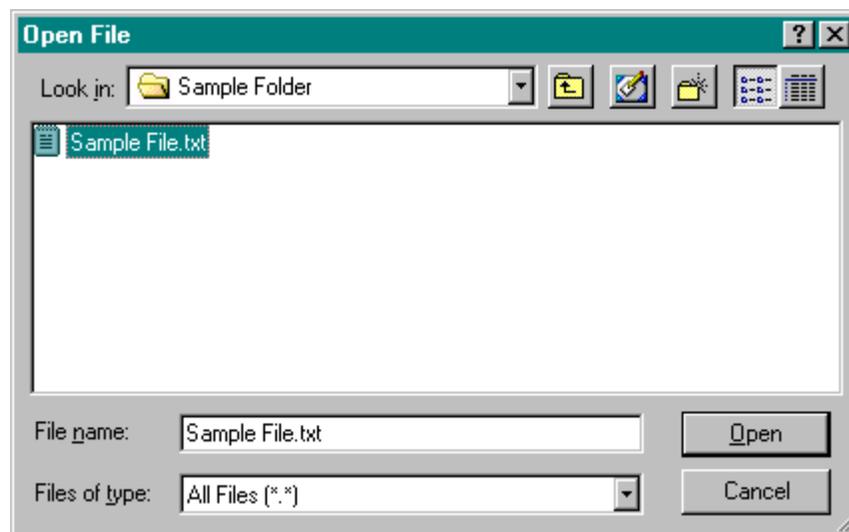


Figure 9.3.3: A standard file | open Dialog box

¹⁴ The equivalent Swing component is `JFileChooser`. Sample code to use that class would be:

```
JFileChooser chooser = new JFileChooser();
if (chooser.showOpenDialog(parent) == JFileChooser.APPROVE_OPTION)
  System.out.println("You chose: " + chooser.getSelectedFile().toString());
```

If a `FileDialog` is instantiated using the `SAVE` constant, it will also block until a user either cancels the dialog or selects or types a file name and clicks on `Save`. If a file is selected for writing that already exists the dialog will automatically ask the user for confirmation to overwrite the existing file. An application can use `getFile` to check for `null` to determine whether the user canceled the dialog. If `getFile` does not return `null` the user has either selected a new file name¹⁵ or confirmed overwriting an existing file. Figure 9.3.4 shows a standard "File | Save" dialog box for the Windows 98 operating system:

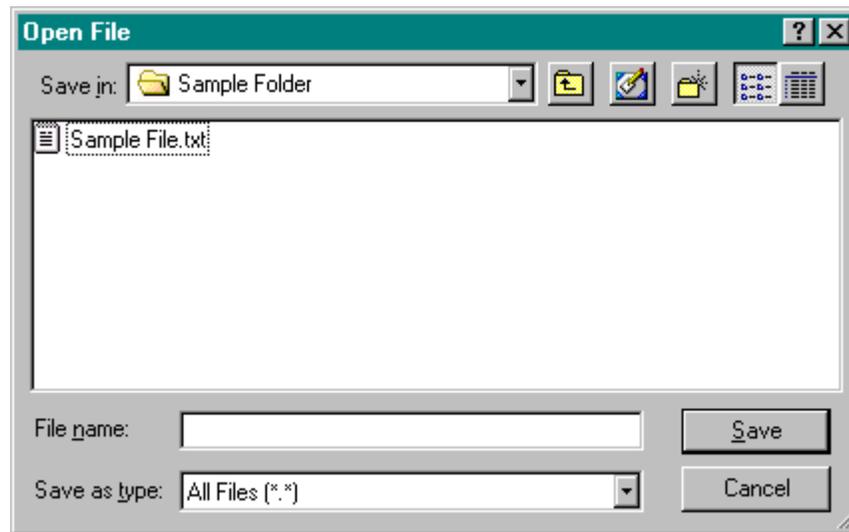


Figure 9.3.4: A standard file | save Dialog box

Now we have all the ingredients to create our program. You should recall how to implement menu bars and menu choices from *The Menu, MenuBar, and MenuItem Classes* in section 4.6.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class TextViewer extends Frame implements ActionListener
{ private Menu fileMenu = new Menu("File");
  private MenuItem fileOpen = new MenuItem("Open");
  private MenuItem fileExit = new MenuItem("Exit");
  private TextArea text = new TextArea();
  public TextViewer()
  { super("Text Viewer");
    fileMenu.add(fileOpen); fileOpen.addActionListener(this);
    fileMenu.addSeparator();
    fileMenu.add(fileExit); fileExit.addActionListener(this);
    MenuBar menu = new MenuBar();
    menu.add(fileMenu);
    setMenuBar(menu);
    setLayout(new BorderLayout());
    add("Center", text);
    text.setEditable(false);
    setSize(400,400);
    setVisible(true);
  }
}
```

¹⁵ The file name may or may not be valid. Appropriate error catching or methods of the `File` class should be used to determine if a file name is valid.

```
public void readFile(String file)
{   text.setText("");
    try
    {   BufferedReader in = new BufferedReader(new FileReader(file));
        String line;
        while ((line = in.readLine()) != null)
            text.append(line + "\n");
        in.close();
        text.setCaretPosition(0);
    }
    catch(IOException ioe)
    {   System.err.println(ioe); }
}
public void actionPerformed(ActionEvent ae)
{   if (ae.getSource() == fileExit)
        System.exit(0);
    else if (ae.getSource() == fileOpen)
    {   FileDialog fd =new FileDialog(this,"Open File",FileDialog.LOAD);
        fd.setVisible(true);
        if (fd.getFile() != null)
        {   File file = new File(fd.getDirectory() + fd.getFile());
            if (file.exists())
                readFile(file.toString());
            else
                text.setText("File name: " + file + " invalid.");
        }
        fd.dispose();
    }
}
public static void main(String args[])
{   TextViewer editor = new TextViewer(); }
}
```

Setting up the layout for the frame is straightforward. During construction the `TextArea` is marked as not editable because we do not want to give the user the impression that this program would allow any changes to the file. The `readFile` method reads data from the text file one line at a time and appends each line to the `TextArea`.¹⁶ After entering all lines into the `TextArea`, the cursor is positioned at the beginning of the `TextArea`. In the `actionPerformed` method, a standard `FileDialog` is instantiated in `LOAD` mode when the user selects the `Open` menu item. When the dialog closes and the method `getFile` does not return `null`, the user has selected a file to open. Since the file may be in another directory, both the `getFile` and `getDirectory` methods are used to construct the complete file name. Finally we make sure that the selected file really exists then pass the name into the `readFile` method to read the file.¹⁷ Figure 9.3.5 shows a picture of the program containing its own source code:

¹⁶ This is a rather slow way to display the file in a `TextArea`. A faster method would be to first generate a `String` containing all characters of the file, then use the `setText` method to swap the entire text into the `TextArea` all at once.

¹⁷ Because of the 32KB restriction on a `TextArea` this program will crash without warning if a file is read that is larger than 32KB. You could use the `length` method of the `File` class to check the file size before attempting to read it. The Swing component `JTextArea` has no such restriction.

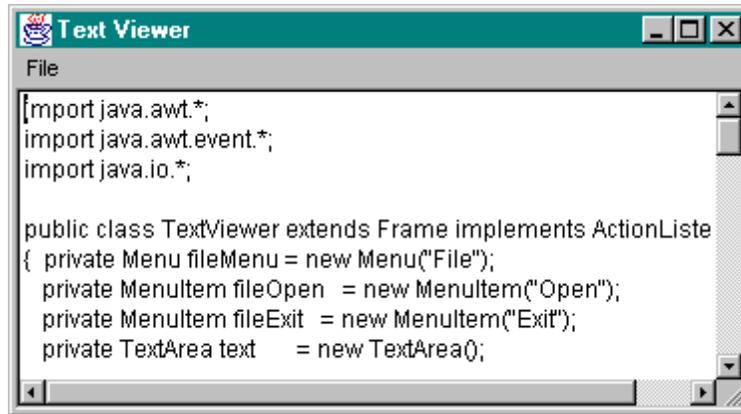


Figure 9.3.5: The `TextViewer` program showing its own source code

There is at least one conceptual flaw in our text viewer program: when the user chooses a file in a different directory, the program will not remember that directory for subsequent opens, contrary to what users might expect. Every time a user opens a file, the dialog will start in the initial directory, not in the directory containing the last file opened. You should rectify that situation as an exercise.

Example 9.3.6:

Create a class that constructs and displays a bar chart from a data file. The data files that are compatible with this class must be plain text files containing one number per line. All numbers should be treated as doubles, even if they do not contain a decimal point. The class should produce a suitable error message if it can not find the input data file or if the file contains characters not representing a double value. Make sure to test your class in every situation.

This exercise presents us with two problems:

- how do we read data values representing double numbers without using a `DataInputStream`
- how do we store the numbers for later processing if we do not know how many numbers will be in the file

To solve the first problem, we will use the `readLine` method of the `BufferedReader`, then use the `doubleValue` method of the `Double` class to attempt to convert the number into a `Double`. To solve the second problem, we will use a `LinkedList`¹⁸, which can grow dynamically as needed. Here is an outline of our class that extends `Canvas` so that it can easily be used in different situations:

```
import java.awt.*;
import java.util.*;
import java.io.*;

public class BarChart extends Canvas
{ private LinkedList data = new LinkedList();
  private double min, max;
  public BarChart(String fileName)
  { /* calls readData to read data file, initialize data, min,
    and max */ }
  private void readData(String file)
  { /* reads data file via BufferedReader, converts numbers to
    Double objects, inserts them into the data list, and
```

¹⁸ See definition 8.3.2.

```

        finds the smallest and largest data values */ }
    private int scalePoints(Object numberObject)
    { /* rescales data values so that they are integers between 0
      and the height of the canvas */ }
    public void paint(Graphics g)
    { /* draws the various bars, using the scaled data points for
      the height */ }
}

```

Using this approach, however, we can not take care of possible errors. The `paint` method will be called regardless of whether an error occurred while attempting to read and convert the data file. Therefore, we add two fields to our class so that `readData` can communicate any possible errors to the `paint` method. In the `paint` method we will either draw the bars or display an appropriate error message.

```

public class BarChart extends Canvas
{ private static String ERRORS[] = {"File not found", "Invalid data"};
  private int errorNumber = -1;
  private LinkedList data = new LinkedList();
  private double min, max;
  public BarChart(String fileName)
  private void readData(String file) throws IOException
  { /* reads data file via BufferedReader, converts numbers to
    Double objects, inserts them into the data list, and
    finds the smallest and largest data values.
    If an error occurring while converting the data values, sets
    errorNumber to the appropriate integer. If an error occurs while
    opening or reading the data, it is passed up to the calling
    method. */ }
  private int scalePoints(Object numberObject)
  public void paint(Graphics g)
  { /* if (errorNumber < 0), draws the various bars, otherwise draw
    an error message string using the above static messages. */ }
}

```

Implementing the constructor is simple: it will just call `readData`, setting `errorNumber` to 0 or 1 if an error occurs:

```

public BarChart(String fileName)
{ try
  { readData(fileName); }
  catch(IOException ie)
  { errorNumber = 0; }
  catch(NumberFormatException nfe)
  { errorNumber = 1; }
}

```

The `readData` method, on the other hand, will do the actual work of reading the data, converting it to `Double` objects if possible and inserting them into the data list. Note that we *must* use `Double` objects instead of `double` values because a `LinkedList` can contain only objects, not basic data types. The method will also check if the number just converted is bigger or smaller than the current `max` and `min`, resetting these values if necessary. If an error occurs during number conversion, it automatically throws a `NumberFormatException`. If an error occurs while opening or reading the data file, it automatically throws an `IOException`. Both exceptions are caught by the constructor and converted into appropriate error codes:

```

private void readData(String file)
  throws IOException, NumberFormatException
{ max = -Double.MAX_VALUE;

```

```

min = Double.MAX_VALUE;
String line;
BufferedReader reader = new BufferedReader(new FileReader(file));
while ( (line = reader.readLine()) != null)
{
    Double number = Double.valueOf(line);
    if (number.doubleValue() > max)
        max = number.doubleValue();
    if (number.doubleValue() < min)
        min = number.doubleValue();
    data.add(number);
}
reader.close();
}

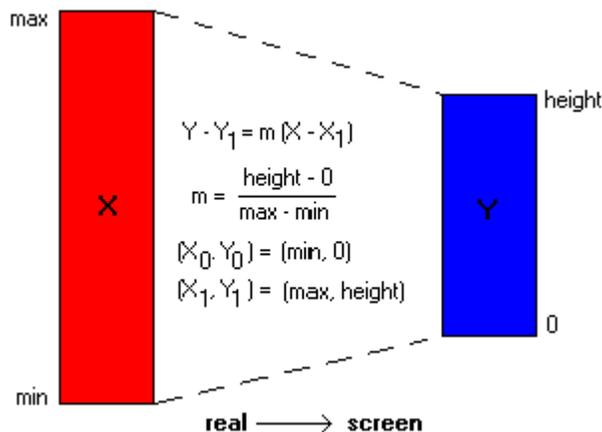
```

Now that the data has been inserted in the data list, we implement the `scalePoint` method to scale a number so that it is between 0 and the height of the screen. Since we know the minimum and maximum value of the data, the scaling operation is a simple computation¹⁹:

```

private int scalePoint(Object numberObject)
{
    double num = ((Double)numberObject).doubleValue();
    return (int)Math.round((num - min)/(max - min) * (getSize().height));
}

```



The familiar point-slope formula for the equation of a line gives the formula that scales the data values so that they fit inside the screen coordinates 0 to `getSize().height`.

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

or

$$y - 0 = \frac{\text{height} - 0}{\text{max} - \text{min}}(x - \text{min})$$

Figure 9.3.6: Scaling data values to screen values

Finally, the `paint` method will check whether an error has occurred by testing `errorNumber`. If no error occurred, it will draw the bars. Otherwise it will draw an appropriate error message:

```

public void paint(Graphics g)
{
    if (errorNumber < 0)
    {
        int width = getSize().width / data.size();
        int counter = 0;
        for (Iterator iterator = data.iterator(); iterator.hasNext(); )20
        {
            int height = scalePoint(iterator.next());
            int x = counter * width, y = getSize().height - height;
            g.setColor(Color.red);    g.fillRect(x, y, width-3, height);
            g.setColor(Color.black);  g.drawRect(x, y, width-3, height);
            counter++;
        }
    }
    else

```

¹⁹ Compare to the `toWorld` method of the Mandelbrot Set example in chapter 6, example 6.6.4.

²⁰ See definition 8.3.16 for details on using an `Iterator`.

```

        g.drawString(ERRORS[errorNumber],10, getSize().height / 2);
    }

```

The basic `BarChart` class is now complete but it could use numerous improvements that we will tackle in the exercises. For now, we can create a simple class to test it:

```

import java.awt.*;
import java.awt.event.*;

public class BarChartMain extends Frame
{
    private class WindowCloser extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    }
    public BarChartMain()
    {
        super("BarChart Test");
        add(new BarChart("sample.dat"));
        setSize(400,400);
        setVisible(true);
        addWindowListener(new WindowCloser());
    }
    public static void main(String args[])
    {
        BarChartMain chart = new BarChartMain();
    }
}

```

Figure 9.3.7 shows some screen shots of our class in action for various data sets:



Figure 9.3.7: BarChart with (a) valid data, (b) non-existing data file, and (c) invalid data

Saving Character Data

Now that we understand how to create character-based streams for reading we need to discuss how to write text to a stream or file. Three classes work together to allow convenient buffered character-based output to a file.

Definition 9.3.7: The `BufferedWriter` Class

The `BufferedWriter` class is used to create a buffered character output stream. Its main use is to provide a buffered character stream to other classes such as the `PrintWriter` class. The Java API defines `BufferedWriter` class as follows:

```

public class BufferedWriter extends Writer
{
    // selected constructor
}

```

```
public BufferedWriter(Writer out)
// selected methods
public void write(int c) throws IOException
public void newLine() throws IOException
public void flush() throws IOException
public void close() throws IOException
}
```

This class does not provide convenient methods to write data other than characters. However, it can - and should - be used in conjunction with the next class, `PrintWriter`:

Definition 9.3.8: The `PrintWriter` Class

The `PrintWriter` class provides convenient methods for formatted, character-based output to a stream. While the `PrintWriter` class could be used by itself, it is best used in conjunction with a `BufferedWriter` class to improve the efficiency of the stream. The methods contained in this class do not throw any exceptions. Instead, the `checkError` method can be used to inquire whether any operation prior to this call has resulted in an error. The Java API defines `PrintWriter` as follows:

```
public class PrintWriter extends Writer
{ // selected constructors
  public PrintWriter(Writer out)
  public PrintWriter(Writer out, boolean autoFlush)
  // selected methods
  public void flush()
  public void close()
  public boolean checkError()
  protected void setError()
  public void print(boolean b)
  public void print(char c)
  public void print(int i)
  public void print(double d)
  public void print(String s)
  public void println()
}
```

The methods writing the basic data types are also available in a `println` version to write the specified data type followed by a line break appropriate for the operating system.

The last class necessary before we can start some examples is the class that ties a character output stream to a file:

Definition 9.3.9: The `FileWriter` Class

The `FileWriter` class connects a file to a character output stream. The Java API defines `FileWriter` as follows:

```
public class FileWriter extends OutputStreamWriter
{ // selected constructors
  public FileWriter(File file) throws IOException
  public FileWriter(String fileName) throws IOException
  public FileWriter(String fileName, boolean append)
    throws IOException
}
```

Note that `OutputStreamWriter` extends `Writer` so that a successful construction of a `FileWriter` object will return an object of type `Writer`.

WARNING: A `FileWriter` will delete an existing file without warning. You can use methods provided by the `File` or `FileDialog` class to ensure that you do not accidentally erase an important file.

Often the three classes `FileWriter`, `BufferedWriter`, and `PrintWriter` are used together to create a buffered text-based output stream to a file where the `PrintWriter` will provide the most functional methods.

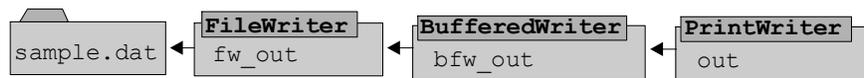


Figure 9.3.8: Instantiating a buffered `FileWriter` stream using a `PrintWriter`

Example 9.3.10:

Create a class that uses a `PrintWriter` to write a double, an int, a char, a boolean, and a `String` value to an output file. When done, open the output file with a standard text editor. Is it readable using a standard text editor? Would the output file still be readable if you used a `DataOutputStream` instead of a `PrintWriter`?

Our first task is to use a `PrintWriter` to create an appropriate output file. We can model our code after example 9.2.1:

```

import java.io.*;

public class SimpleCharOutputTest
{
    public static void main(String args[])
    {
        double Pi = 3.1415;
        int i = 10;
        boolean okay = true;
        char cc = 'J';
        String s = "Java by Definition";
        try
        {
            FileWriter fw = new FileWriter("sample-char.dat");
            PrintWriter out = new PrintWriter(new BufferedWriter(fw));
            out.println(Pi);
            out.println(i);
            out.println(okay);
            out.println(cc);
            out.println(s);
            out.close();
            if (out.checkError())
                System.out.println("An error has occurred during output.");
        }
        catch(IOException ioe)
        {
            System.out.println("Error while opening the file.");
        }
    }
}
  
```

When the program is executed, it will create the data file "sample-char.dat". We can open that file with a standard text editor such as Notepad and the result will look similar to figure 9.3.9

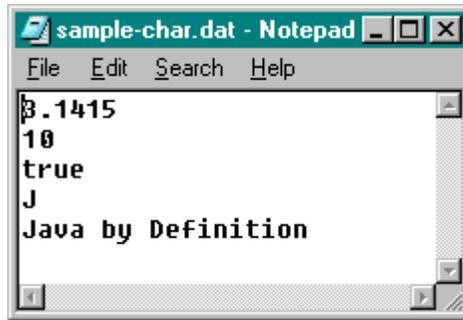


Figure 9.3.9: Using Notepad to look at a character-level output file

The data is perfectly readable because it was created by a character-based stream. If we execute the program "SimpleOutputTest" that we created in example 9.2.1, using a `DataOutputStream` instead of a `PrintWriter` to write the data, we can also open the resulting data file with Notepad. However, it will look similar to figure 9.3.10:



Figure 9.3.10: Using Notepad to look at a byte-level output file

The only part of the data we can read is the character and the string. The rest consists of "funny" characters. Of course the data file can be read by a `DataInputStream`, but *only* by that class. Files created with a `PrintWriter` can be read by a `BufferedReader` but also by other programs. ■

A Simple Text Editor Program

As our final example in this section we will create a complete text editor program that can load, edit, and save text files.

Example 9.3.11:

Create a simple text editor program similar to the Notepad application that comes with Windows. The program should be able to create new files, retrieve existing files, view and modify text, and save files as text files. It does not have to print files and it does not need to worry about "automatic line wrapping".

This is a significant task, not really because of the saving and retrieving features that our program should provide, but because we need to make sure that we create a program that does not accidentally overwrite existing files, or lose data that has just been edited. In fact, the actual editing and displaying of text is handled automatically by a `TextArea`. The only problem we need to be concerned with is saving and retrieving the data at the right time. First, let's outline the class by figuring out the fields and methods it should contain.

Clearly, the class representing the program should extend `Frame` and implement `ActionListener`. As for the fields:

- we need various menu items that should be combined in a `File` menu and added to the menu bar of the frame
- we need a `TextArea` to contain the editable text and actually, the `TextArea` automatically handles the adding and modification of text²¹
- we need to store the name of the file and directory where the file is located

As usual, the fields are easy (but we actually will need at least one more field, introduced later). In this case, however, the methods are also easy. After all, everyone has used a text editor program, or word processing program, and it should be completely clear *what* such a program needs to accomplish:

- we need a method `newFile` to create a new, empty file
- we need a method `openFile` to open an existing file and display it in the `TextArea`
- we need a method `saveFile` to save the existing file, using the existing file name if one has been chosen already (otherwise, the method should call `saveFileAs` instead)
- we need a method `saveFileAs` to save the text under a new name
- since both `saveFile` and `saveFileAs` will ultimately write data to disk, we add the method `savingFile` to actually handle the saving, given a file name and directory
- we need an exit method that only allows the user to exit if either all text modifications are saved or the user has been explicitly warned that changes may be lost

Finally, our class or program needs to display dialog boxes to inform or ask the user. For example, if the user chooses `File | Exit` without having saved the file, the program must display a dialog asking the user whether to save the current file or not. The same is true if the user wants to open a file without having first saved the current one.

- Therefore, we add one more method `wantToSave` to our class.

Figure 9.3.11 shows an outline of our program so far:

```

Editor
extends Frame
implements ActionListener

MenuItems fileNew, fileOpen, fileSave
           fileSaveAs, fileExit;
TextAreas text;
String file;

Editor()
void exitProgram();
void saveFile();
void saveFileAs();
void newFile();
void openFile();
void actionPerformed(ActionEvent ae);
boolean wantToSave();
void savingFile();
void main(String args[]);

```

Figure 9.3.11: Representation of the `Editor` program

```
import java.awt.*;
```

²¹ That's the good news. The bad news is that as soon as a `TextArea` forms the basis of our program, it automatically is restricted by the 32KB size limitation of that component.

```

import java.awt.event.*;
import java.io.*;

public class Editor extends Frame implements ActionListener
{
    private MenuItem fileNew      = new MenuItem("New");
    private MenuItem fileOpen     = new MenuItem("Open ...");
    private MenuItem fileSave     = new MenuItem("Save");
    private MenuItem fileSaveAs   = new MenuItem("Save As ...");
    private MenuItem fileExit     = new MenuItem("Exit");
    private TextArea text         = new TextArea();
    private String file = "untitled";

    public Editor()                { }
    public void exitProgram()      { }
    public void saveFile()         { }
    public void saveFileAs()      { }
    public void newFile()         { }
    public void openFile()        { }
    public void actionPerformed(ActionEvent ae) { }
    private boolean wantToSave()  { }
    private void savingFile()     { }
    public static void main(String args[]) { }
}

```

Before we start implementing the various methods, however, we have one big problem: while the `TextArea` class handles the actual editing of the file by itself, our `Editor` class needs to know exactly when the text displayed in the `TextArea` changes. After all, if the displayed text changes we must ask the user to save the file before letting them open a new file or exit the program. On the other hand, if the text did not change we can let the program exit without warning the user to save the file.

That feat is accomplished by attaching a `TextListener` to the `TextArea`. Using that listener our class can find out exactly when the text is changing. A class implementing `TextListener` needs to provide the method `void textValueChanged(TextEvent te)` which is called, as the name implies, when the value of a text component changes. To distinguish files that need saving from files whose contents has not changed, we use an additional field of type `boolean` called `isDirty` and `textValueChanged` will set `isDirty` to `true`. Hence, the complete outline of our class looks as in figure 9.3.12:

```

Editor
    extends Frame
    implements ActionListener, TextListener

    MenuItems fileNew, fileOpen, fileSave
               fileSaveAs, fileExit;
    TextAreas text;
    String file;
    boolean isDirty;

    Editor()
    void exitProgram();
    void saveFile();
    void saveFileAs();
    void newFile();
    void openFile();
    void actionPerformed(ActionEvent ae);
    void textValueChanged(TextEvent te);
    boolean wantToSave();
    void savingFile();
    void main(String args[]);

```

Figure 9.3.12: Representation of modified `Editor` program

```

import java.awt.*;
import java.awt.event.*;

```

```

import java.io.*;

public class Editor extends Frame implements ActionListener, TextListener
{
    private MenuItem fileNew      = new MenuItem("New");
    private MenuItem fileOpen     = new MenuItem("Open ...");
    private MenuItem fileSave     = new MenuItem("Save");
    private MenuItem fileSaveAs  = new MenuItem("Save As ...");
    private MenuItem fileExit     = new MenuItem("Exit");
    private TextArea text        = new TextArea();
    private String file = "untitled";
    private boolean isDirty = false;

    public Editor()
    public void exitProgram()
    public void saveFile()
    public void saveFileAs()
    public void newFile()
    public void openFile()
    public void textValueChanged(TextEvent te)
    public void actionPerformed(ActionEvent ae)
    private boolean wantToSave()
    private void savingFile()
    public static void main(String args[])
}

```

With this outline in place, it should be easy to implement the various methods. The constructor simply builds the menu and sets an initial size for the frame:

```

public Editor()
{
    super("Text Viewer");
    setTitle("Editor: untitled");
    Menu fileMenu = new Menu("File");
    fileMenu.add(fileNew);      fileNew.addActionListener(this);
    fileMenu.add(fileOpen);    fileOpen.addActionListener(this);
    fileMenu.addSeparator();
    fileMenu.add(fileSave);    fileSave.addActionListener(this);
    fileMenu.add(fileSaveAs);  fileSaveAs.addActionListener(this);
    fileMenu.addSeparator();
    fileMenu.add(fileExit);    fileExit.addActionListener(this);
    text.addTextListener(this);
    MenuBar menu = new MenuBar();
    menu.add(fileMenu);
    setMenuBar(menu);
    setLayout(new BorderLayout());
    add("Center", text);
    setSize(400,400);
    setVisible(true);
}

```

The `saveFile` method checks the variable `file` to see if a file name has been determined already. If the file name is "untitled" it calls `saveFileAs`, otherwise it calls `savingFile` to save the file under its current name. In either case, the method sets `isDirty` to false to indicate that so far the file in memory is the same as the version on disk.

```

public void saveFile()
{
    if (file.equals("untitled"))
        saveFileAs();
    else
        savingFile();
    isDirty = false;
}

```

The `saveFileAs` method brings up a `FileDialog` in `SAVE` mode to prompt the user for a file name. Remember that if the user picks a file that already exists the file dialog asks for confirmation whether to overwrite that file or not. Therefore, we do not need to check again whether the file that is returned by the `FileDialog` already exists. Also, we use both the `getDirectory` and the `getFile` method to get the complete path and file name for the file to be saved. The actual saving is done via the `savingFile` method. The method also sets `isDirty` to false to indicate that so far the file in memory is the same as the version on disk.

```
public void saveFileAs()
{
    FileDialog fd = new FileDialog(this, "Save File As", FileDialog.SAVE);
    fd.show();
    if (fd.getFile() != null)
    {
        file = fd.getDirectory() + fd.getFile();
        savingFile();
    }
    isDirty = false;
}
```

The `newFile` method first checks whether the file in memory is different from the one on disk. If so, it calls on the method `wantToSave` to ask the user whether to save the current file before proceeding. Note that `wantToSave` will only execute if `isDirty` is true. Then the `TextArea` is cleared, the file's title is set to "untitled", and `isDirty` is set to false.

```
public void newFile()
{
    if ((isDirty) && (wantToSave()))
        saveFile();
    text.setText("");
    file = "untitled";
    setTitle("Editor: " + file);
    isDirty = false;
}
```

The `openFile` method again first checks if the file is dirty. If so, it asks the user whether to save the current file, and saves it if desired. Then the method uses a `FileDialog` to prompt the user for a file to open. If the user picks a valid file, it reads the text via a `BufferedReader` into a temporary string variable `s`, appending a "new line" character after each line. When the entire text file is read into `s`, the string's content is moved into the text field and the cursor is set to the top of the text. Finally, `isDirty` is set to false to indicate that so far the text in memory is identical to the text on disk.

```
public void openFile()
{
    if ((isDirty) && (wantToSave()))
        saveFile();
    FileDialog fd = new FileDialog(this, "Open File", FileDialog.LOAD);
    fd.show();
    if (fd.getFile() != null)
    {
        file = fd.getDirectory() + fd.getFile();
        setTitle("Editor: " + file);
        try
        {
            BufferedReader in = new BufferedReader(new FileReader(file));
            String line = new String("");
            while ((line = in.readLine()) != null)
                s +=line + "\n";
            in.close();
            text.setText(s);
            text.setCaretPosition(0);
        }
        catch(IOException ioe)
        {}
    }
}
```

```
        { System.err.println(ioe); }
        isDirty = false;
    }
    fd.dispose();
}
```

The `exitProgram` method is simple: it checks whether the file is dirty and if so whether the user wants to save it. If yes, the file is saved via the `saveFile` method, and in any case the `System.exit(0)` method terminates the program.

```
public void exitProgram()
{ if ((isDirty) && (wantToSave()))
    saveFile();
  System.exit(0);
}
```

The method `wantToSave` uses the `YesNo` class we designed in example 4.37 to bring up an appropriate dialog asking the user whether to save the current file. You must make sure that `YesNo.java` is located in the same directory as the `Editor.java` file before this method will compile. `YesNo` in turn needs the class `ButtonRow` from example 4.36 to be present in the current directory before it compiles. The method eventually returns `true` or `false`, depending on whether the user clicked the first or second button.

```
private boolean wantToSave()
{ YesNo saving = new YesNo(this, "File not Saved",
                           "Do you want to save " + file, "Yes", "No");
  return saving.okay();
}
```

The method `savingFile` simply saves the entire text in the `TextArea` using a `PrintWriter`. Since the text in the `TextArea` already contains line breaks, they will be written to disk as well. That means, however, that if the underlying platform the program is running on is Windows/DOS, the lines are not properly terminated with carriage return *and* line feed. To fix this, we will refer you to the exercises.

```
private void savingFile()
{ try
  { PrintWriter out = new PrintWriter(new FileWriter(file));
    out.print(text.getText());
    out.close();
  }
  catch(IOException ioe)
  { System.out.println(ioe); }
  setTitle("Editor: " + file);
}
```

The `textValueChanged` method is called automatically every time the text in the `TextArea` changes. If that happens, the text in memory will be different from any version on disk, so that we should consider the `TextArea` as "dirty". Hence, the only thing this method does is to set `isDirty` to `true`.

```
public void textValueChanged(TextEvent te)
{ isDirty = true; }
```

We could have also used an anonymous inner class for this listener. The `actionPerformed` method simply calls the appropriate methods when the user picks a menu item. The actual work is done by the above methods.

```

public void actionPerformed(ActionEvent ae)
{
    if (ae.getSource() == fileExit)
        exitProgram();
    else if (ae.getSource() == fileNew)
        newFile();
    else if (ae.getSource() == fileOpen)
        openFile();
    else if (ae.getSource() == fileSave)
        saveFile();
    else if (ae.getSource() == fileSaveAs)
        saveFileAs();
}

```

Finally, the main method instantiates an `Editor` object to set the program in motion.

```

public static void main(String args[])
{
    Editor editor = new Editor();
}

```

All pieces put together should make a simple but usable basic text editor program.²² Other features such as cut-and-paste, automatic line wrapping, etc. can be added by you.

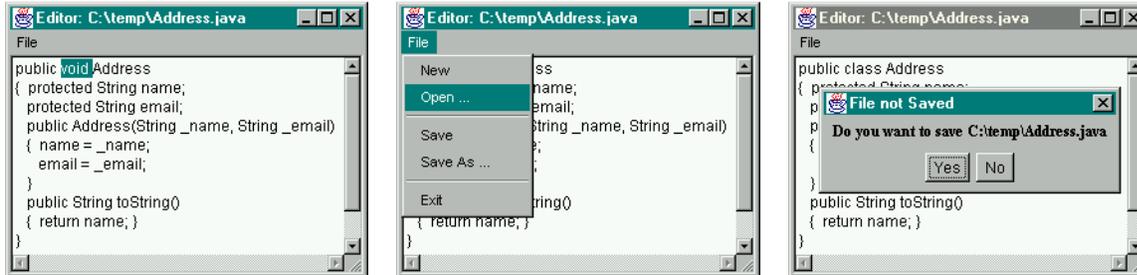


Figure 9.3.13: Various stages of the `Editor` program

8.4. Saving and Retrieving Object Data

So far we have investigated classes that can save and retrieve basic data types and strings. If we wanted to save an entire object used to store data, we would have to create a method that saves the individual fields of the object, and another to read the various fields back from disk to reconstitute the object's state. It would be nice if we could save the entire state of the object in one swoop, storing and retrieving field values automatically. Moreover, we do not have any possibility to save any of the objects that Java provides, such as `Button`, `Label`, or `Canvas`. We would have to save basic data types that define the state of such an object via "hand-made" methods and then restore the state of such an object by reading the basic types back.

Example 9.4.1:

Suppose we are working on an address book program that handles two types of addresses: one type stores name (first name only) and email address, the other stores

²² As we mentioned, the main restriction of our program is the 32KB limit imposed by the `TextArea`. The easiest way to overcome that is to convert the program to Swing. The biggest problem doing that would be to replace the `TextListener` by an appropriate `DocumentListener`. See definition 6.3.18.

first name, email, and phone number. Create a simple `AddressBook` program that has the following features:

- it can add a random address of either type
- it can display the addresses in a `List`
- it can clear all addresses
- it can save and restore the addresses currently in the list

Let's first create the basic `Address` and `AddressPhone` classes. Each class should have the capability of setting and storing the respective data, converting its data to a string, as well as send and receive its data through a stream. As the type of stream we choose a `DataInputStream` and a `DataOutputStream` because the data file that will ultimately be created by our `AddressBook` program will only be used by that program. The implementation of each class is simple, so here is the `Address` class:

```
Address
String first, email;
Address()
Address(String _first, String _email);
String toString();
void save(DataOutputStream out) throws IOException;
void load(DataInputStream in) throws IOException;
```

Figure 9.4.1: Representation of Address class

```
import java.io.*;

public class Address
{   protected String first, email;
    public Address()
    {   first = email = ""; }
    public Address(String _first, String _email)
    {   first = _first;
        email = _email;
    }
    public String toString()
    {   return first + " (" + email + ")"; }
    public void save(DataOutputStream out) throws IOException
    {   out.writeUTF(first);
        out.writeUTF(email);
    }
    public void load(DataInputStream in) throws IOException
    {   first = in.readUTF();
        email = in.readUTF();
    }
}
```

The `AddressPhone` class will extend `Address` and override these methods with its own versions:

```
AddressPhone   extends Address
String phone;
Address()
Address(String _first, String _email, String _phone);
String toString();
void save(DataOutputStream out) throws IOException;
void load(DataInputStream in) throws IOException;
```

Figure 9.4.2: Representation of the AddressBook class

```

import java.io.*;

public class AddressPhone extends Address
{
    protected String phone;
    public AddressPhone()
    { first = email = phone = ""; }
    public AddressPhone(String _first, String _email, String _phone)
    { super(_first, _email);
      phone = _phone;
    }
    public String toString()
    { return super.toString() + " - " + phone; }
    public void save(DataOutputStream out) throws IOException
    { super.save(out);
      out.writeUTF(phone);
    }
    public void load(DataInputStream in) throws IOException
    { super.load(in);
      phone = in.readUTF();
    }
}

```

In particular, the `save` and `load` methods are inverse of each other: `save` writes the values of the fields using a `DataOutputStream` while `load` reads the values back in exactly the same order as `save` wrote them, using a `DataInputStream`.

Now let's create the actual `AddressBook` class. We clearly need an assortment of buttons for the `add`, `clear`, `load`, `save`, and `quit` operations. We also need a structure to store the addresses and a structure to display them. We will use a `LinkedList` for storing addresses and a `List` (from `java.awt`) to display them. We will use one additional field holding a particular `Address` and a specific `AddressPhone`. When the user chooses to add an address, we select at random one of these addresses and add them to the linked list and the `List`. The methods that our class needs are as follows:

- the constructor defines the layout of the GUI elements
- `addData` will use an `Address` as input and add it to the `List` as well as the `LinkedList`
- `clearData` will clear the `LinkedList` to be empty and remove all addresses from the `List`
- `saveData` will create an appropriate `DataOutputStream` and save all addresses
- `loadData` will create an appropriate `DataInputStream` and retrieve all previously saved addresses
- `actionPerformed` will delegate the work according to which button was pressed

Here is the implementation of the class except for `loadData` and `saveData`. As it turns out, we can not use the type `List` (from `java.awt`) to declare the type of the display field. Because we are importing all classes from `java.awt` *and* from `java.util`, we have a class and an interface that are both called `List`. Therefore, we need to use the full name of the class, including its package name, to specify exactly which `List` we mean:

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class AddressBookTest extends Frame implements ActionListener
{
    private Button add = new Button("Add");
    private Button clear = new Button("Clear");
}

```

```

private Button save = new Button("Save");
private Button load = new Button("Load");
private Button quit = new Button("Quit");
private java.awt.List display = new java.awt.List(7);
private LinkedList data = new LinkedList();
private Address[] ADRS ={ new Address("Bert","bert@x.y"),
                          new AddressPhone("jbd","jbd@x.y","1234")};

public AddressBookTest()
{ Panel buttons = new Panel();
  buttons.setLayout(new FlowLayout());
  buttons.add(add); add.addActionListener(this);
  buttons.add(clear); clear.addActionListener(this);
  buttons.add(save); save.addActionListener(this);
  buttons.add(load); load.addActionListener(this);
  buttons.add(quit); quit.addActionListener(this);
  setLayout(new BorderLayout());
  add("Center", display);
  add("South", buttons);
  validate(); pack(); setVisible(true);
}

public void saveData()
{ /* to be implemented later */ }

public void loadData()
{ /* to be implemented later */ }

public void addData(Address address)
{ data.add(address);
  display.add(address.toString());
}

public void clearData()
{ data.clear();
  display.removeAll();
}

public void actionPerformed(ActionEvent ae)
{ if (ae.getSource() == quit)
    System.exit(0);
  else if (ae.getSource() == add)
    addData(ADRS[(int)(ADRS.length*Math.random())]);
  else if (ae.getSource() == save)
    saveData();
  else if (ae.getSource() == load)
    loadData();
  else if (ae.getSource() == clear)
    clearData();
}

public static void main(String args[])
{ AddressBookTest ab = new AddressBookTest(); }
}

```

The class will already compile and execute, but of course the saving and retrieving mechanisms do not yet work. To implement them, the idea is simple:

- For `saveData`, we will iterate through the elements of the linked list, type-casting each object as an `Address`, and call on its `save` method. Polymorphism will select the particular `save` method appropriate to the actual type of the object. Imagining what the `loadData` method will do to restore the data, we also save the size of the current linked list before actually writing any address data. The resulting data file could look as in figure 9.4.3 where we saved 3 addresses (an `Address`, and `AddressPhone`, and another `Address`)

- For `loadData`, we start with reading the first integer (see line one in figure 9.4.3). That number will indicate how many addresses are stored in the file. Now we could use a `for` loop to read in the various strings and creating appropriate `Address` or `AddressPhone` objects. However, our problem is that we do not know whether a particular group of strings saved to disk represents an `Address` or an `AddressPhone`. We can not use polymorphism here because all data saved is of type `String` and contains no reference to the actual type of object that originally contained the data. Theoretically, for example, lines 2, 3, and 4 could be an `AddressPhone` object and the remaining lines two `Address` objects.



Figure 9.4.3: Address data²³

Therefore, we need to modify the `save` method of `Address` and `AddressPhone` so that it saves the type of object in addition to the values of the fields of the object. Using that information, we can later decide whether to reconstitute an `Address` or an `AddressPhone`. The `save` method of `Address` should be replaced by:

```
public void save(DataOutputStream out) throws IOException
{
    out.writeUTF("Address");
    out.writeUTF(first);
    out.writeUTF(email);
}
```

That means that the `save` method of `AddressPhone` can no longer use its superclass method and instead must be implemented as follows:

```
public void save(DataOutputStream out) throws IOException
{
    out.writeUTF("AddressPhone");
    out.writeUTF(first);
    out.writeUTF(email);
    out.writeUTF(phone);
}
```

The `load` methods of both classes should remain unchanged, because we need to read the string indicating the type of address *before* calling the respective `load` method.

With these redefined methods in place we can implement the `saveData` and `loadData` methods of the `AddressBook`:

```
public void saveData()
{
    try
    {
        DataOutputStream out = new DataOutputStream(
            new FileOutputStream("addresses.dat"));
        out.writeInt(data.size());
        for (Iterator iterator = data.iterator(); iterator.hasNext(); )
            ((Address)iterator.next()).save(out);
        out.close();
    }
    catch (IOException ioe)
    {
        System.out.println(ioe);
    }
}
```

²³ The actual `save` method writes strings through a byte-level stream. Therefore figure 9.4.3 only represents the idea of the data not an actual view of the saved data.

```
    }
```

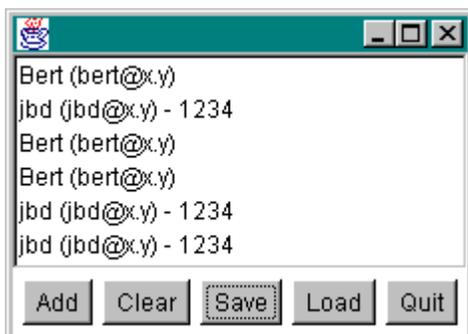
This method is just as described earlier: first it writes the number of addresses in the linked list to the stream, then it iterates through the elements in the linked list, typecasts each object into an `Address`, and uses whichever `save` method is selected via polymorphism to write the data to the stream. Our modified `save` methods will store the appropriate type of the address as a string before writing the values of the fields for each object.

Here is the `loadData` method: It first reads the integer indicating the number of addresses stored. Then, in a `for` loop it reads a string and decides based on that string which type of address is stored in the data. It correspondingly instantiates either an `Address` or an `AddressPhone` object and uses its respective `load` method to read the appropriate number of fields.

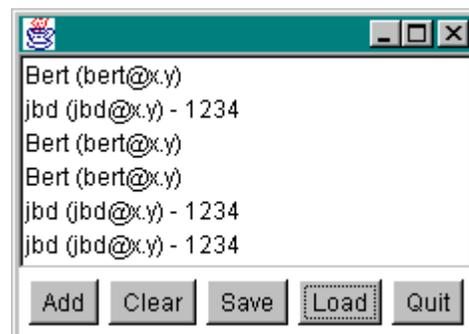
```
public void loadData()
{ try
  { DataInputStream in = new DataInputStream(
                                new FileInputStream("addresses.dat"));

    clearData();
    int counter = in.readInt();
    for (int i=0; i < counter; i++)
    { String type = in.readUTF();
      if (type.equals("Address"))
      { Address address = new Address();
        address.load(in);
        addData(address);
      }
      else if (type.equals("AddressPhone"))
      { AddressPhone address = new AddressPhone();
        address.load(in);
        addData(address);
      }
    }
    in.close();
  }
  catch(IOException ioe)
  { System.out.println(ioe); }
}
```

When all methods are put in their appropriate place, the program should run fine and save and retrieve addresses in the appropriate format as necessary (see figure 9.4.4):



AddressBookTest with addresses saved to disk



AddressBookTest after loading saved data

Figure 9.4.4: AddressBookText program in action

In the left picture of figure 9.4.4 we have added a few addresses to the list. You can see that the two types of addresses show up at random, and polymorphism has selected the appropriate `toString` method. The data is then saved and the list cleared (which, of course, you can not see). On the right side we have clicked on the `Load` button to load the data back from disk. As you can see, the load method did correctly reconstitute the `Address` and `AddressBook` objects, just as they were originally, and polymorphism again can select the corresponding `toString` method. ■

While the program in the previous example certainly works well, it is not elegant: the `loadData` method does not use polymorphism, one of the basic advantages of object oriented programming, and it seems cumbersome to save and retrieve the data fields "by hand". It would be much easier if objects could be endowed with a standard universal mechanism that would save the entire state of the object to a stream, regardless of the number of fields and the type of the object.

Saving Object Data

Indeed, Java does provide facilities to transfer objects through streams as a whole. That mechanism does not work for all objects, but it works particularly well for objects whose main function is that of data storage and retrieval such as the above `Address` and `AddressBook` classes. Two Java classes and an Interface work together to enable object saving and retrieving. The first class is responsible for writing entire objects to an output stream:

Definition 9.4.2: The `ObjectOutputStream` Class

An `ObjectOutputStream` is used to write primitive data types as well as entire objects to a stream. In order to be written to an `ObjectOutputStream`, objects must implement the `java.io.Serializable` interface. The default mechanism for writing an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields. The Java API defines this class as follows:

```
public class ObjectOutputStream extends OutputStream
    implements ObjectOutput, ObjectOutputStreamConstants
{
    // selected constructor
    public ObjectOutputStream(OutputStream out) throws IOException
    // selected methods
    public final void writeObject(Object obj) throws IOException
    public void close() throws IOException
    public void writeBoolean(boolean data) throws IOException
    public void writeByte(int data) throws IOException
    public void writeChar(int data) throws IOException
    public void writeInt(int data) throws IOException
    public void writeDouble(double data) throws IOException
    public void writeChars(String data) throws IOException
    public void writeUTF(String data) throws IOException
}
```

As mentioned in the definition, only objects that implement the `Serializable` interface can be written via an `ObjectOutputStream`. Fortunately, that interface is particularly simple:

Definition 9.4.3: The Serializable Interface

Classes that want to be serializable must implement the `Serializable` interface. In our situation this means that classes implementing `Serializable` indicate their willingness to be written to an `ObjectOutputStream` and read via an `ObjectInputStream`. The interface is particularly simple because it does not contain any methods that must be implemented.

All subclasses of a serializable class are themselves serializable. The `Serializable` interface is part of the `java.io` package.

Many basic Java classes already implement `Serializable`, such as for example the `Date` class. Other classes, particularly "hand-made" classes, need to explicitly implement that interface if desired. Once a class is marked as serializable, objects constructed from it can be written to an output stream.

Example 9.4.4:

Create a simple program that saves an integer, a date, and an `Address` to an `ObjectOutputStream`.

Simple in this case means that we not need to be concerned with GUI elements, we can concentrate on writing the data to the stream. Our first task is to make sure that the `Address` class from our previous example becomes serializable. We therefore must make sure that the `Address` class implements `Serializable` (we also remove the `save` and `load` methods from the `Address` class because they are no longer necessary²⁴):

```
import java.io.*;

public class Address implements Serializable
{   protected String first, email;
    public Address()
    {   first = email = ""; }
    public Address(String _first, String _email)
    {   first = _first;
        email = _email;
    }
    public String toString()
    {   return first + " (" + email + ")"; }
}
```

The class is a lot simpler than our previous version. In particular, if we add or remove fields from the class we do not need to modify any special "save" or "load" methods. Implementing `Serializable` is free, we do not have to add any new methods.²⁵

The `Date` class, by definition, is already serializable so that we can proceed with our example:

```
import java.io.*;
import java.util.*;

public class SerialWriteTest
{   public static void main(String args[])
    {   try
```

²⁴ We could of course keep these methods. After all, they were marked `public` and it is usually not a good idea to change public methods in a class because other programs may rely on those methods.

²⁵ There really is no such thing as a free lunch., and there is of course a downside to serializing objects, which we will mention in definition 9.4.8.

```
    { ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("sample.dat"));
      int i = 10;
      Date now = new Date();
      Address address = new Address("Bert", "Bert@x.y");
      out.writeInt(i);
      out.writeObject(now);
      out.writeObject(address);
      out.close();
    }
    catch(IOException ioe)
    { System.out.println(ioe); }
  }
}
```

When this class executes, it will create the `sample.dat` data file containing the necessary information to restore, or deserialize, the appropriate objects and integer. ■

Of course we will only know whether the `SerialWriteTest` program worked correctly if we can successfully retrieve the objects that were saved.

Retrieving Object Data

As with the previous output stream classes, an `ObjectOutputStream` also has its appropriate counterpart:

Definition 9.4.5: The `ObjectInputStream` Class

An `ObjectInputStream` is used to read primitive data types as well as entire objects from a stream that have previously been written via an `ObjectOutputStream`. Only objects that support the `java.io.Serializable` interface can be read from streams. Java's type casting should be used to restore the object to their proper type. The Java API defines this class as follows:

```
public class ObjectInputStream extends InputStream implements
    ObjectInput, ObjectStreamConstants
{ // selected constructor
  public ObjectInputStream(InputStream in) throws IOException,
    StreamCorruptedException

  // selected methods
  public final Object readObject() throws OptionalDataException,
    ClassNotFoundException, IOException
  public void close() throws IOException
  public boolean readBoolean() throws IOException
  public byte readByte() throws IOException
  public char readChar() throws IOException
  public int readInt() throws IOException
  public double readDouble() throws IOException
  public String readUTF() throws IOException
}
```

Using this class we can test whether our previous example really worked:

Example 9.4.6:

In the previous example we created a class to write an integer, a `Date` object and an `Address` object to a file. Create a class that reads that data back and prints it to the screen for verification.

To reconstitute, or deserialize, objects we need to instantiate an `ObjectInputStream` and use the `readObject` method together with type-casting. The `readObject` method can throw several exceptions we need to catch, but only `IOException` and `ClassNotFoundException` *must* be caught. The `OptionalDataException` extends `IOException` and can be caught together with that exception. The object input stream also provides methods to read the basic data types, so we can use `readInt` to read the integer back:

```
import java.io.*;
import java.util.*;

public class SerialReadTest
{   public static void main(String args[])
    {   try
        {   ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("sample.dat"));

            int i = in.readInt();
            Date date = (Date)in.readObject();
            Address address = (Address)in.readObject();
            in.close();
            System.out.println("Integer: " + i);
            System.out.println("Date: " + date);
            System.out.println("Address: " + address);
        }
        catch(ClassNotFoundException cnfe)
        {   System.out.println(cnfe); }
        catch(IOException ioe)
        {   System.out.println(ioe); }
    }
}
```

The class should compile without problem as long as the `Address` class from example 9.4.4 is available. It will retrieve the original objects just as they were saved (see figure 9.4.5).

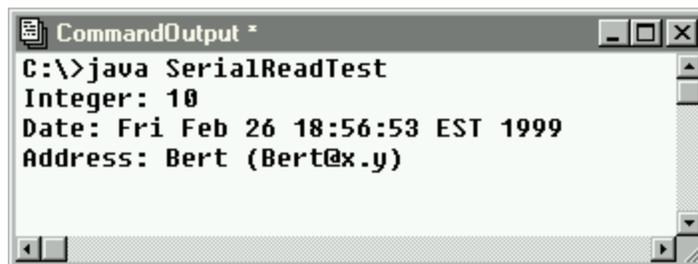


Figure 9.4.5: Retrieving objects via an `ObjectInputStream`

Note that the `SerialReadTest` class only works because we know the exact type of objects and basic types that were saved originally, as well as the order in which they were saved. Without that information - which is *not* part of the file - it would be much more difficult to try to read `sample.dat`.

Serializing objects is a very convenient mechanism and it greatly simplifies the transfer of objects through streams. Most importantly, the advantages of object oriented programming such as

polymorphism can be used again without having to resort to "hand-made" solutions as in our original `AddressBookTest` example.

Example 9.4.7:

In example 9.4.1 we created an `Address`, `AddressPhone`, and `AddressBookTest` class to save and retrieve addresses via data input and output streams. Modify these classes so that you can take full advantage of object streams instead of data streams.

Our first task is to modify all objects to be serializable. We already changed the `Address` class in example 9.4.6:

```
import java.io.*;

public class Address implements Serializable
{   protected String first, email;
    public Address()
    {   first = email = ""; }
    public Address(String _first, String _email)
    {   first = _first;
        email = _email;
    }
    public String toString()
    {   return first + " (" + email + ")"; }
}
```

We do not need the original `save` and `load` methods at all (we could have left them in the class, though, just in case). The new `AddressPhone` class is even simpler:

```
import java.io.*;

public class AddressPhone extends Address
{   protected String phone;
    public AddressPhone()
    {   first = email = phone = ""; }
    public AddressPhone(String _first, String _email, String _phone)
    {   super(_first, _email);
        phone = _phone;
    }
    public String toString()
    {   return super.toString() + " - " + phone; }
}
```

We do not even have to explicitly implement the `Serializable` interface because the class extends the serializable class `Address` and is therefore automatically serializable. All we did is delete the `save` and `load` methods that we no longer need.

The modifications in the `AddressBookTest` class only impact on the `saveData` and `loadData` methods, everything else remains exactly the same as before. Here is the structure of that class (see example 9.4.1 for details):

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class AddressBookTest extends Frame implements ActionListener
{   /* Buttons: add, clear, save, load, quit
    List: display (from java.awt)
```

```

        LinkedList: data
    */
    public AddressBookTest()
    { /* layout all components and make frame visible */ }
    public void saveData()
    { /* new implementation, see below */ }
    public void loadData()
    { /* new implementation, see below */ }
    public void clearData()
    { /* removes data from display and data list */ }
    public void actionPerformed(ActionEvent ae)
    { /* decides when to do what */ }
    public static void main(String args[])
    { /* instantiates an instance of itself */ }
}

```

The `saveData` method used to write the number of elements in the data list to a stream, then iterated through the data list. Each element was type-cast to `Address` and saved via the appropriate `save` method. Our first attempt of improving the data transfer will be similar, but using an `ObjectOutputStream` instead of a `DataOutputStream`. No type-casting will be necessary, and - our most important improvement - the entire object state is saved via the `writeObject` method without having to call on specific methods of the object:

```

    public void saveData()
    { try
      { ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("addresses.dat"));
        out.writeInt(data.size());
        for (Iterator iterator = data.iterator(); iterator.hasNext(); )
          out.writeObject(iterator.next());
        out.close();
      }
      catch(IOException ioe)
      { System.err.println(ioe); }
    }

```

The method looks similar to the previous version, but we do no longer need to resort to methods like `save` specific to the particular object to be written to the stream. Instead, we write the entire state of the object to the stream.

The `loadData` method is significantly simplified over its previous version because each object saved automatically includes its exact type. Therefore, when objects are read back polymorphism can be used perfectly fine:

```

    public void loadData()
    { try
      { ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("addresses.dat"));
        int size = in.readInt();
        clearData();
        for (int i = 0; i < size; i++)
          addData((Address)in.readObject());
        in.close();
      }
      catch(ClassNotFoundException cnfe)
      { System.out.println(cnfe); }
      catch(IOException ioe)
      { System.out.println(ioe); }
    }

```

The new version of `AddressBookTest` works exactly the same as the previous one from the user's perspective. However, from the programmer's perspective it is simpler and more elegant (but we will find an even better solution below): when an object is transferred through the `DataInputStream`, it is type-cast as `Address`. However, that object knows exactly what type it is, and if it is of type `AddressPhone` it will preserve that identity (of course objects of type `AddressPhone` are *also* of type `Address`). Thus, when adding the address to the lists, polymorphism can select the appropriate `toString` method, as before.

Actually, we kind of stopped too soon without realizing the full potential of being able to transfer complete object: why not transfer the entire list through the object stream instead of transferring each element separately? Indeed, that's what object streams are particularly good for: transferring data objects, i.e. objects whose main use is to store data. Since a linked list is a quintessential data object, we should save and retrieve it in one piece. That would also eliminate the need to store the size of the list first, because the list itself, when written to an `ObjectOutputStream`, will continue to remember its size.

Therefore, we will once again modify the `saveData` and `loadData` as follows:

```
public void saveData()
{ try
  { ObjectOutputStream out = new ObjectOutputStream(
                                new FileOutputStream("addresses.dat"));
    out.writeObject(data);
    out.close();
  }
  catch(IOException ioe)
  { System.err.println(ioe); }
}

public void loadData()
{ try
  { ObjectInputStream in = new ObjectInputStream(
                                new FileInputStream("addresses.dat"));
    clearData();
    data = new LinkedList();
    data = (LinkedList)in.readObject();
    in.close();
  }
  catch(ClassNotFoundException cnfe)
  { System.out.println(cnfe); }
  catch(IOException ioe)
  { System.out.println(ioe); }
  for (Iterator iterator = data.iterator(); iterator.hasNext(); )
    display.add(((Address)iterator.next()).toString());
}
```

The new `saveData` is just about as simple as can be: it writes the entire list to the stream and leaves all details to serialization. The `loadData` method, at first glance, still seems to iterate through the elements in the list, but not to read them from the `ObjectInputStream`. Instead, *first* the entire list is read via `readObject`, then we iterate through the data list to add the appropriate strings to the `java.awt.List display`.²⁶

²⁶ We no longer need to call the `addData` method. In fact, we *should not* call that method any more because after `readObject` is done the entire list is restored. The `addData` method wants to add addresses *again* to the list, which would no longer be necessary.

While serialization can be *very* convenient, it also introduces a security problem. The `Date` class, for example, makes it virtually impossible to construct or set an invalid date. The constructor and its corresponding methods will check for the validity of a date and refuse invalid entries. However, if a `Date` object is written to disk through an `ObjectOutputStream`, someone could manipulate the bytes in the output file so that they represent an invalid date. Since the data exists on disk independent of the attached object, nothing can protect the data from being changed. When the `Date` object is read back it could represent an invalid date, compromising the security of the `Date` class. The same is true with private field values: inside the protection of an object they can not be manipulated from the outside, but once written to disk they are exposed to external manipulations by anyone (with sufficient skills).

Definition 9.4.8: Rule of Thumb for Protecting Objects during Serialization

Serialized objects that were written to disk exist without the protection that Java offers objects stored inside a JVM. If the security of objects is at a premium you may not want to allow default serialization of that object. To customize serialization, add the methods:

```
public void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
{ /* customize the way objects are read */ }
public void writeObject(ObjectOutputStream out)
    throws IOException
{ /* customize the way objects are written */ }
```

To completely prevent an object from being serialized, add the method:

```
public void writeObject(ObjectOutputStream out)
    throws IOException
{ throw NoAccessException(); }
```

The Serial in `Serializable`: The "Jane meets Jimmy" Example

While there may be a potential security risk with saving serializable objects, there is a tremendous benefit that we have not yet mentioned. Serializing, or saving objects via object streams, allows us to preserve data that has internal relationships with each other. The next example will illustrate that, as well as explain the name "serializable" for this particular interface.

Example 9.4.8:

In this example we want to model a relationship as it evolves between Jane, Joe, and Jimmy. First, Jane and Joe do not know each other. Then they meet and become friends. A while later they part ways and Jane meets Joe who replaces Jimmy in their relationship. Write a program that describes this little story:

- without saving and retrieving data
- saving and retrieving data before the relationship changes, using character-level streams
- saving and retrieving data before the relationship changes, using object-level streams
-

Explain any problems that you find.

To model this story, we can use several methods:

- a "Jane meets Joe" method, where the relationship between Jane and Joe is setup
- a "Jimmy replaces Joe" method, where Jimmy replaces Joe in the relationship
- a "show Relations" method to show the current state of affairs

Of course we first need a class to describe a relationship, which is easy. We create the `Relation` class where a relation consists of a person with a name who has one relation (for simplicity):

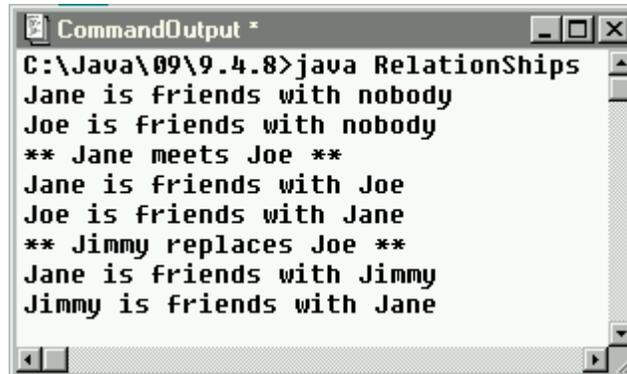
```
public class Relation
{   private String name;
    private Relation friend;
    public Relation(String _name)
    {   name = _name; }
    public void setName(String _name)
    {   name = _name; }
    public String getName()
    {   return name; }
    public Relation getFriend()
    {   return friend; }
    public void setFriend(Relation _friend)
    {   friend = _friend; }
    public String getFriendName()
    {   if (friend == null)
        return "nobody";
        else
        return friend.getName();
    }
}
```

Now we create the main class to model our story: we setup an array of two relations, one for Jane and one for Joe. Initially, they have no relationship with each other. Then we create methods `JaneMeetsJoe` and `JimmyReplacesJoe` as well as a main method to tell the story:

```
public class Relationships
{   public static Relation friends[] =
        {new Relation("Jane"), new Relation("Joe")};
    public static void JaneMeetsJoe()
    {   System.out.println("** Jane meets Joe **");
        friends[0].setFriend(friends[1]);
        friends[1].setFriend(friends[0]);
    }
    public static void JimmyReplacesJoe()
    {   System.out.println("** Jimmy replaces Joe **");
        friends[1].setName("Jimmy");
    }
    public static void showRelationships()
    {   for( int i = 0; i < friends.length; i++)
        System.out.println(friends[i].getName() +
                            " is friends with " +
                            friends[i].getFriendName());
    }
    public static void main(String args[])
    {   showRelationships();
        JaneMeetsJoe();
        showRelationships();
        JimmyReplacesJoe();
        showRelationships();
    }
}
```

```
    }
}
```

Now we can tell the story of this relationship by simply executing the `Relationships` programs, as shown in figure 9.4.6:



```

C:\Java\09\9.4.8>java Relationships
Jane is friends with nobody
Joe is friends with nobody
** Jane meets Joe **
Jane is friends with Joe
Joe is friends with Jane
** Jimmy replaces Joe **
Jane is friends with Jimmy
Jimmy is friends with Jane

```

Figure 9.4.6: Relationships without saving/retrieving

Everything is as it should. Next, we add the "save/retrieve" feature to our program (maybe because the original relationship between Jane and Joe lasts quite a while before she meets Jimmy). If we want to use text files to store the data, we add appropriate save and load methods to the `Relation` class to handle the actual saving and retrieving of the data:

```
import java.io.*;

public class Relation
{ /* fields and methods as before, plus these two additional methods: */

    public void save(PrintWriter out) throws IOException
    { out.println(name);
      out.println(friend.getName());
    }

    public void load(BufferedReader in) throws IOException
    { name = in.readLine();
      friend = new Relation(in.readLine());
    }
}
```

The `save` method saves the name of the person with the relation as well as the name of the person they are related to. The `load` method retrieves the relation's name, and constructs a new relation to the original friend's name. We then change the main method of the `Relationships` class to save and retrieve the data before Jimmy replaces Joe:

```
import java.io.*;

public class Relationships
{ /* everything stays the same, but the main method changes: */
    public static void main(String args[])
    { showRelationships();
      JaneMeetsJoe();
      showRelationships();

      try
      { PrintWriter out = new PrintWriter(
          new FileWriter("friends.dat"));
        friends[0].save(out); friends[1].save(out);
      }
    }
}
```

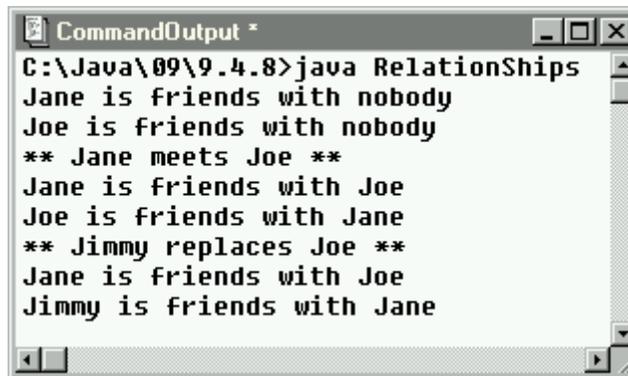
```

        out.close();
        BufferedReader in = new BufferedReader(
            new FileReader("friends.dat"));
        friends[0].load(in); friends[1].load(in);
        in.close();
    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }

    JimmyReplacesJoe();
    showRelationships();
}
}

```

Now we execute this modified program, expecting exactly the same output as before. After all, we simply saved and retrieved the data without modifying the actual story line. But figure 9.4.7 shows a different situation:



```

C:\Java\09\9.4.8>java Relationships
Jane is friends with nobody
Joe is friends with nobody
** Jane meets Joe **
Jane is friends with Joe
Joe is friends with Jane
** Jimmy replaces Joe **
Jane is friends with Joe
Jimmy is friends with Jane

```

Figure 9.4.7: Relationships changed when saving/retrieving to text file

Jane and Joe were friends, as before, but when Jimmy replaces Joe, Joe continues to linger in Jane's memory, even though according to our program Jimmy should have replaced Joe entirely. The problem must be introduced while saving/retrieving the data. Before we explore what actually happened, let's change our code to use object streams instead of character streams.

The only modification to the `Relation` class is that it now must implement `Serializable`:

```

import java.io.*;

public class Relation implements Serializable
{ /* no further changes (save and load methods could be removed) */ }

```

The main method of `Relationships` gets easier, because we can save the entire array without using a loop:

```

import java.io.*;

public class Relationships
{ /* everything stays the same, but the main method changes: */
    public static void main(String args[])
    { showRelationships();
      JaneMeetsJoe();
      showRelationships();

      try
      { try

```

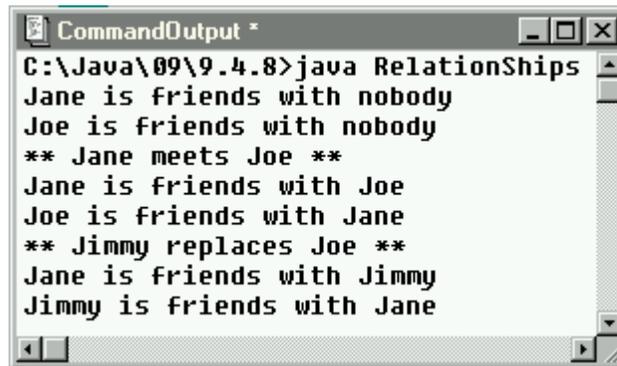
```

    { ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("friends.obj"));
      out.writeObject(friends);
      out.close();
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("friends.obj"));
      friends = (Relation[])in.readObject();
      in.close();
    }
  catch(Exception ex)
  { System.err.println("Error: " + ex); }

  JimmyReplacesJoe();
  showRelationships();
}
}

```

Note that we must catch an `IOException` as well as a `ClassCastException`, which we do by catching the superclass `Exception`. When we execute this last modification of `Relationships`, figure 9.4.8 shows that everything is back to normal again even though we save and retrieve the state of affairs before Jimmy appears:



```

C:\Java\09\9.4.8>java Relationships
Jane is friends with nobody
Joe is friends with nobody
** Jane meets Joe **
Jane is friends with Joe
Joe is friends with Jane
** Jimmy replaces Joe **
Jane is friends with Jimmy
Jimmy is friends with Jane

```

Figure 9.4.8: Relationships saved and retrieved correctly via object streams

Now let's explore what is going wrong when saving data via character streams and how serializing can avoid the problem. The class `Relation` has two fields, a `String` and another `Relation` field. When we save the data as text, we save the `String` part of a `Relation` just fine as text, but we can not save the second field of type `Relation` directly. Therefore we save only the name of the `Relation` field, which is again of type `String`. There is little else for us to do if all we can save is character data. The data saved, therefore, is:

```

Jane
Joe
Joe
Jane

```

indicating that the first `Relation` has name `Jane` and a friend named `Joe`, and the second `Relation` is named `Joe` with a friend named `Jane`. What is not reflected in this saved data is that both occurrences of `Joe` are referring to the *same* object in memory, as shown in figure 9.4.9:

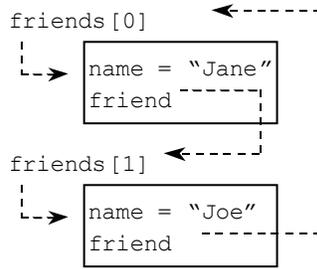


Figure 9.4.9: Relation objects in memory before saving

When the data is retrieved from the text file, a Relation named Jane is created, which has a new Relation named Joe as friend. A second Relation named Joe is then created, which is different from the friend Joe of the first relation. This new Joe will get as friend another Relation named Jane. Now there are *four* objects of type Relation in memory, whereas we really only meant to save and retrieve two objects. The objects in memory after retrieval are shown in figure 9.4.10.

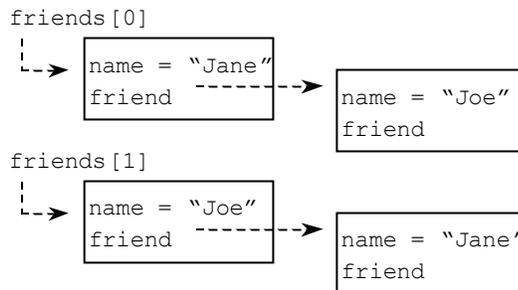


Figure 9.4.10: Objects in memory after retrieving

When we then change the name of friends[1] to "Jimmy" it has no impact on the name of the Relation that the friend field of friends[0] points to.

When we use object streams, serialization is able to avoid that problem by assigning each object a unique serial number (hence the name serialization). When data is saved, the serial number of the objects in memory is also saved, so that retrieval can use the serial numbers to correctly restore the relationships between the original objects, as indicated in figure 9.4.11.

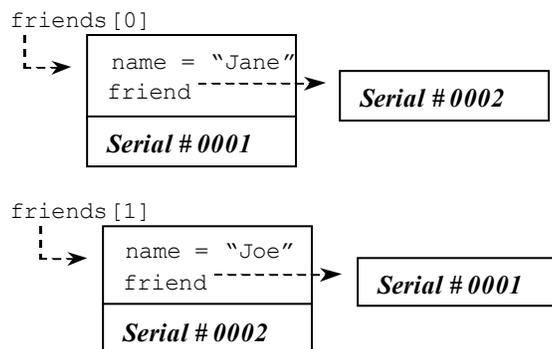


Figure 9.4.11: Objects with serial numbers showing their relationships

The data that is saved to disk would reflect that the `friend` field of object with serial number 0001 refers to the object with serial number 0002, while the `friend` field of object 0002 refers to object 0001. Now the correct relationship between these objects can be restored after retrieving them.²⁷ ■

Up to this point, all examples dealing with reading or writing files were stand-alone programs, not applet. That was not an oversight. Instead, applets are *not permitted* to read or write files at all by the default security manager that governs the behavior of all applets. The next section will investigate security restrictions imposed on applets. Since many of these restrictions are related to reading and writing files, the section is included in this chapter even though no fundamentally new techniques for reading and writing will be introduced.

8.5. Security, URL's, and Applets

As we mentioned in chapter 1, before a Java class executes it is scrutinized for any security problems or violations. Even while running, every Java class is controlled by a security manager that governs access to all resources that are available to the class. If a class requests access to a resource it should not be able to get, the security manager denies the request and throws a `SecurityException` instead. With Java 1.1 and especially 1.2, the programmer has the ability of creating "trust-worthy" classes such that the user of the applet can have some control over what resources should be available to the class. The user can choose to be less restrictive than the default security manager and grant access to resources that would otherwise be unavailable to the class. The latest version of Java, in fact, allows a lot of flexibility when dealing with different types of security managers and even lets you create your own customized security manager.²⁸

Security Restrictions for Applets

We will investigate some of the default security restrictions that an applet has to adhere to, and what some of the standard solutions are to overcome these restrictions in a safe manner. Our first example will illustrate one of the basic problems for applets trying to deal with file input/output:

Example 9.5.1:

Create two applets: one that writes a single string to a text file, and another that opens a text file containing one line and reads the line. Run the applets using (a) the appletviewer, (b) using your web browsers loading the applet locally, and (c) using your web browser loading the applet from an actual web site. Describe the results. You should open the Java console window on your web browser before running the applet.²⁹

²⁷ The problem is introduced because the `save` method of `Relation` saves the `friend` name as a `String`, not as an object (which it can't). But one could try to call on the `save` method of the `friend` field itself, to attempt to avoid the problem. However, that will not solve anything but rather end up in an infinite loop. See the exercises for details.

²⁸ A customized security manager can not replace the one used by a web browser. Applets running in a web browser are controlled by that browser's security manager.

²⁹ For Netscape Communicator 4.5, choose `Communicator | Tools | Java Console` from the menu. For Internet Explorer, you must first select `View | Internet Options ...` from the menu, then click on the `Advanced` tab to

The code for both applets is simple: the first applet gets a single `Write` button. When clicked, we use a `PrintWriter` attached to a `FileWriter` to write a string to file.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class WriteAppletTest extends Applet implements ActionListener
{
    private Button write = new Button("Write");
    public void init()
    {
        setLayout(new FlowLayout());
        add(write); write.addActionListener(this);
    }
    public void writeFile()
    {
        try
        {
            PrintWriter out = new PrintWriter(new FileWriter("sample.dat"));
            out.println("Java by Definition");
            out.close();
        }
        catch(IOException ioe)
        {
            System.out.println(ioe);
        }
    }
    public void actionPerformed(ActionEvent ae)
    {
        if (ae.getSource() == write)
            writeFile();
    }
}
```

When this applet is started (via the appropriate `HTML` file) using the `appletviewer`, we get the following response when clicking on the button:



Figure 9.5.1: *SecurityException for an Applet trying to write to a file*

In other words, our applet is not allowed to write to file when executed under the `appletviewer` program. If we start the same program from Netscape or Internet Explorer, loading the `HTML` file from the local disk, we get a similar error message. And if we place the file on a web site and then try to click on the `Write` button, again an error is produced and no file is created.

As for reading data from file, the problem is slightly different. First, the code for the applet is simple: it contains a `Read` button and a `Label`. When clicking on `Read` we will use a `BufferedReader` and a `FileReader` to read the string and display it in the label.

```
import java.applet.*;
```

put a check next to the option `Java Console enabled`. After restarting Internet Explorer, you can bring up the Java console by selecting `View | Java Console`.

```

import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class ReadAppletTest extends Applet implements ActionListener
{
    private Button read = new Button("Read");
    private Label display = new Label("Java by Definition");
    public void init()
    {
        setLayout(new FlowLayout());
        add(read); read.addActionListener(this);
        add(display);
    }
    public void readFile()
    {
        try
        {
            BufferedReader in = new BufferedReader(new FileReader("st.dat"));
            display.setText(in.readLine());
            in.close();
        }
        catch(IOException ioe)
        {
            System.out.println(ioe);
        }
    }
    public void actionPerformed(ActionEvent ae)
    {
        if (ae.getSource() == read)
            readFile();
    }
}

```

Before we can test this applet, we of course need to create the data file. Since we are using a Reader, we can create a simple text file (using, say, Notepad) containing the string "it worked fine". The file name for this data file, of course, must be `st.dat`. Make sure the applet code, the corresponding HTML document, and the data file are in the same directory, then run the applet and click on the Read button. Under the appletviewer, our program works fine:



Figure 9.5.2: Reading data from file inside an applet, using the appletviewer

Running the same applet under Netscape Communicator 4.5, again it works fine if we load the HTML file and the applet code from the local disk.³⁰ However, loading the same file from the local disk into Internet Explorer results in a security exception:

³⁰ The situation is even trickier. If we load the applet by selecting `File | Open Page ...`, we can choose the HTML document containing the applet from the local disk and everything works fine. If we then create a bookmark to the same HTML document and load it via the bookmark feature of Netscape, the same applet will *not* be allowed to read data from disk. That discrepancy occurs because the URL's are different in these cases and therefore different security managers will control the applet.

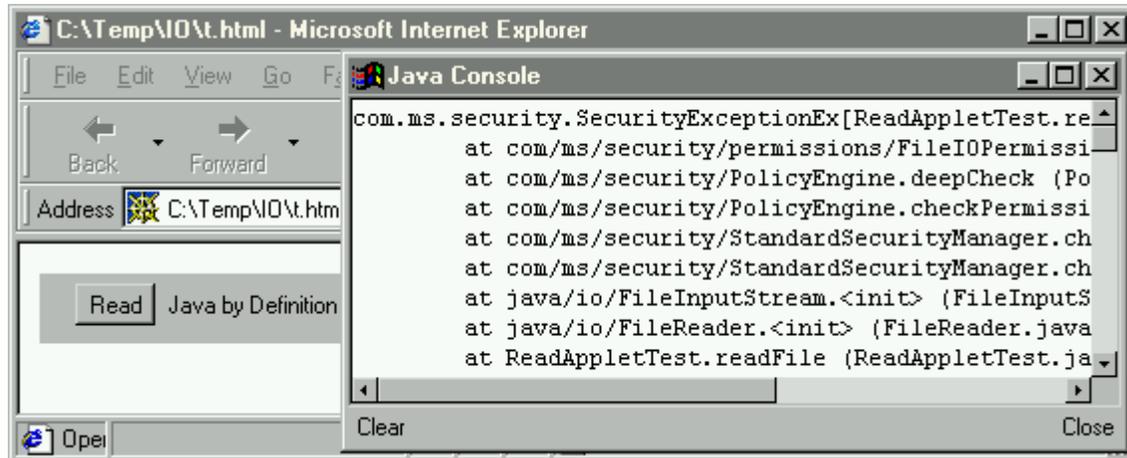


Figure 9.5.3: Security exceptions generated by Internet Explorer when reading from an applet

Both Netscape and Internet Explorer will generate a security exception when running the applet from an actual web site.

To summarize, we have determined the following:

- An applet can not write data to disk using a `FileOutputStream`
- An applet can not consistently read data from disk using a `FileInputStream`

After some thought, this should not be surprising. After all, applets start executing as soon as you access the web page containing it. If an applet were allowed to read data, it could start reading files from your hard disk as soon as you load a particular web page and transmit the contents of your disk back to the creator of the applet. That would probably not be what you want. Or worse, if applets were allowed to write data, an applet could proceed to wipe out all files on your disk as soon as you load the page containing the applet code. That would certainly not be a desirable feature.

Therefore, Java has strong security restrictions that apply by default to applets. Occasionally, some restrictions may not apply when loading an applet using the appletviewer from a local disk. However, you should always make sure to test the applet by loading it from an actual web page to make sure that no security restrictions prevent your applet from doing its work.

The Applet Sandbox

At first it may seem that applets that can - by nature of being applets - neither read nor write data to disk will be very limiting and might induce people not to consider a programming language with these restrictions. However, just the opposite is true: because Java imposes tight security on applets, people feel confident enough that applets can not inflict damage on their systems and are willing to let programs execute automatically on their system. Applets are said to execute in a "sandbox":

Definition 9.5.2: The Applet Sandbox

Applets are said to execute in a Sandbox. That means that applets can perform inside a well-defined area of your computer but they are not allowed to stray outside that area. A security

manager supervises every applet and if it detects that an applet requests access to a resource outside its sandbox, it denies that access and throws a `SecurityException` instead. The particular restrictions that an applet security manager imposes are:

- *Applets can not directly read or write files*
- *Applets can make network connections only to the machine that hosts the applet*
- *Applets can not execute any native programs on the machine it is running on*
- *Applets can not read or modify certain system properties*
- *If an applet brings up a window (such as a frame), the window looks different from standard windows*
- *Applets can not define native methods*

There are, of course, ways to overcome these restrictions. One way is to let the user decide which restrictions should apply to an applet. Java offers the possibility of creating "digitally signed applets". When such an applet starts, a user is presented with a signed security certificate and can choose to relax the default restrictions based on who signed for the applet.

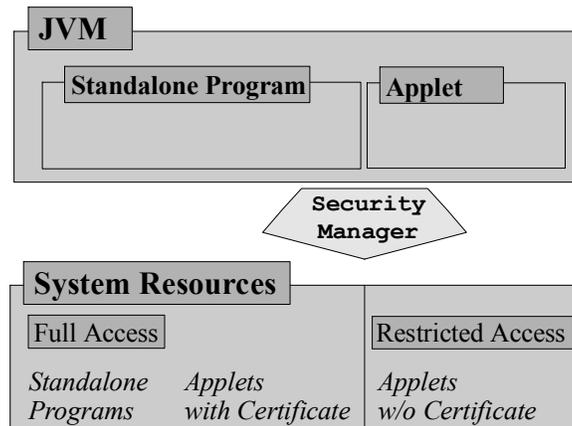


Figure 9.5.4: A security manager controlling access to system resources

For example, users could choose to trust applets from company X, while refusing to grant special privileges to applets from company Y. In fact, Java 1.2 has quite a powerful mechanism to define, implement, and manage digital certificates. However, we will not pursue this further here because users are not likely to grant applets signed by `Jane Student` any special privileges. Instead, we will see how to overcome the restrictions in a secure way using standard programming techniques.

Reading Data from Applets

The restriction we can easily overcome is the one that does not permit applets to read data from disk: applets can actually read data files as long as these data files are explicitly declared as "public files". A "public file" is a file that is located in a publicly readable directory of the same web server that delivered the applet attempting to read the file.

Definition 9.5.3: Rule of Thumb for Reading Data from an Applet

If an applet wants to read data from a file, the following conditions must be met:

- *the data file and the applet must physically be located on the same machine*
- *the data file and the applet must be located either on a local machine, or in the public area of a web server*
- *a connection to the data file must be established using the `URL` class, not a `FileInputStream`*

This rule of thumb applies only to input streams (reading), not to output streams (writing).

To clarify this rule of thumb, here are a few examples (see figures 9.5.5 and 9.5.6):

- If both the applet and the data file are located on drive C: of your computer, the applet can read the data file using the `URL` class
- If both the applet and the data file are located in a personal web account such as, say, `http://www.shu.edu/~wachsmut`, then the applet can read the data file using the `URL` class
- If an applet is located in a web directory on a web server called `http://www.shu.edu/~otheruser`, and the data file is located on the same web server but in a personal web account such as, say, `http://www.shu.edu/~wachsmut`, then the applet can read the data file using the `URL` class



Figure 9.5.5: Locations of Applets and Data Files valid for Reading

- If an applet uses a `FileInputStream`, it will not have access to any file for reading
- If the applet is located on a web server and wants to read a data file located on your computer's hard disk, it will not have access to the file
- If the applet is located on a web server named, for example, `http://www.shu.edu/`, and the data file is located on another web server named, for example, `http://www.cat.shu.edu/`, then the applet will not have access to the file
- If the applet is located on a web server named, for example, `http://www.shu.edu/`, and the data file is located on some account on the same machine but not in a directory accessible by the web server of that machine, then the applet will not have access to the file.



Figure 9.5.6: Locations of Applets and Data Files not valid for Reading

In principle, applets can establish byte (or data) streams, character streams (or readers), or object streams to data files for reading as long as the above conditions are met.

URLs and the URL Class

Before we can start an example, we need to clarify the definition of a URL as well as the definition of the `URL` class in the `java.net` package. Most of the concepts used in that definition will be explained again in chapter 10, section 1.

Definition 9.5.4: Uniform Resource Locator, or URL

A Uniform Resource Locator, or URL, is a pointer to a "resource" on the World Wide Web. A resource can be something as simple as a file or a directory, but could be more complicated such as a reference to a database query. A URL has three distinct parts: the protocol descriptor, the machine address of the host containing the referenced resource with an optional port number, and the location of the resource on the named machine (specific to the protocol and host), usually a file name and location. A full URL is specified using the syntax:

```
protocol://hostname[:port]/location/of/file.name
```

If a port is not specified, the default port for the protocol is used. If protocol and machine name are missing, they are inherited from the resource containing the URL. Such a URL is called partial. If protocol and machine name are missing, and the location of the resource does not start with a slash "/", then the location of the named resource is relative to the location of the resource containing the URL. Such a URL is called relative.

A URL is the underlying concept for the hyperlinks that connect the millions of web pages available on the World Wide Web.

Example 9.5.5:

Suppose a resource such as a web page is located at the location `http://www.shu.edu/stuff/aPage.html`. That web page contains the following URL's, or hyperlinks:

- `http://www.yahoo.com/index.html`
- `/aSecondPage.html`
- `aThirdPage.html`
- `/and/aFourthPage.html`
- `and/aFifthPage.html`
- `ftp://ftp.java.com/somewhere/is/a/file.zip`

Which address is a full URL, which is a partial URL, and which is a relative URL? What are the full URL's corresponding to the partial and relative URL's?

Only the first and last URL have a protocol descriptor and a machine name, so these two are the only full URL's. Both use the default port associated with the protocol descriptor (see chapter 10.1). The second and fourth URL start with a slash, and are therefore partial URL's. The remaining

URL's - third and fifth - are relative URL's. The full URL's of the incomplete addresses are completed using the address of the document containing them:

Base URL of resource: `http://www.shu.edu/stuff/aPage.html`

- partial URL `/aSecondPage.html`
 - completed to `http://www.shu.edu/aSecondPage.html`
- relative URL `aThirdPage.html`
 - completed to `http://www.shu.edu/stuff/aThirdPage.html`
- partial URL `/and/aFourthPage.html`
 - completed to `http://www.shu.edu/and/aFourthPage.html`
- relative URL `and/aFifthPage.html`
 - completed to `http://www.shu.edu/stuff/and/aFifthPage.html`

Closely connected to this concept of a Uniform Resource Locator is the `URL` class of the `java.net` package:

Definition 9.5.6: The `URL` Class

The `URL` class represents a Uniform Resource Locator, or URL. The Java API defines the class as follows:

```
public final class URL extends Object implements Serializable
{ // selected constructors
    public URL(String protocol, String host, int port, String file)
        throws MalformedURLException
    public URL(String protocol, String host, String file)
        throws MalformedURLException
    public URL(URL context, String spec)31
        throws MalformedURLException
    public URL(String spec)
        throws MalformedURLException
    // selected methods
    public int getPort()
    public String getProtocol()
    public String getHost()
    public String getFile()
    public boolean equals(Object obj)
    public String toString()
    public final InputStream openStream() throws IOException
    public final Object getContent() throws IOException
}
```

The method most relevant for our current purpose is the `openStream` method, which can be used to obtain an `InputStream` to read a data file *if* the conditions in the Rule of Thumb for reading data in applets are satisfied.

Example 9.5.7:

Two applets `App1.class` and `App2.class` are assumed to exist on a web server at the URL `http://www.shu.edu/Java/`. The first applet is embedded in an HTML document `http://www.shu.edu/Java/doc1.html` via the `APPLET` tag:

³¹ The most useful technique of providing a context or base URL for this method is via the methods `getCodeBase` and `getDocumentBase` of the `Applet` class. See definition 6.5.9 for details.

```
<APPLET CODE="App1.class" WIDTH="100" HEIGHT="100">
</APPLET>
```

The second applet is embedded in an HTML document located at the address <http://www.shu.edu/~wachsmut/doc2.html> via the `APPLET` tag:

```
<APPLET CODEBASE="http://www.shu.edu/Java" CODE="App2.class"
WIDTH="100" HEIGHT="100">
</APPLET>
```

Both applets want to read the data file <http://www.shu.edu/Java/sample.dat>. What code would be necessary to open an input stream to the data file using the `openStream` method of a URL object? Make the code as flexible as possible.

Our first but not best approach is straightforward: each applet will use a full URL to locate the data file and then use the `openStream` method to open a connection to that file:

```
import java.applet.*;
import java.io.*;
import java.net.*;

public class App1 extends Applet
{ public void init()
  { try
    { URL url = new URL("http://www.shu.edu/Java/sample.dat");
      InputStream in = url.openStream();
      /* do something with the input stream */
      in.close();
    }
    catch (MalformedURLException murle)
    { System.err.println(murle); }
    catch (IOException ioe)
    { System.err.println(ioe); }
  }
}
```

The code for `App2` would be exactly the same. We use a full URL and therefore it does not matter what the location of the applet or underlying HTML document is relative to the data file. However, that code would be restrictive. The applet would now *only* work when it is loaded from the web server <http://www.shu.edu/>. In particular, you could not test your applet locally, because according to the rule of thumb the applet and the data file must be located on the same machine. You could, of course, change the full URL for the data file to refer to a file on your local computer, change it back to the true URL, and transfer the applet to the web server, but that would be rather inconvenient. Another disadvantage is that if the directory `Java` on www.shu.edu was ever renamed or moved elsewhere - including the applet and data file - the applet would no longer work because of the inflexible full URL used.

To help make our applets more flexible, recall the methods `getDocumentBase()` and `getCodeBase()` that every applet inherits, as defined in definition 6.5.9. The method `getDocumentBase()` returns the base address of the HTML document containing the current `APPLET` tag, while `getCodeBase()` returns the base address of the location of the applet code as specified in the `CODEBASE` parameter of the `APPLET` tag. Using these methods we can modify the code for the applets as follows:

```
import java.applet.*;
import java.io.*;
import java.net.*;
```

```
public class App1 extends Applet
{   public void init()
    {   try
        {   URL url = new URL(getDocumentBase(),"sample.dat");
            InputStream in = url.openStream();
            /* do something with the input stream */
            in.close();
        }
        catch(MalformedURLException murle)
        {   System.err.println(murle); }
        catch(IOException ioe)
        {   System.err.println(ioe); }
    }
}
```

Now this applet will work as long as the applet code and the data file `sample.dat` are located in the same directory. The actual name of the directory or the name of the web server will not matter. In particular, you can test the applet locally, as long as the applet and data file are in the same directory, and then transfer both applet and data file to any location on a web server and things would still work correctly.

Since the HTML document for the second applet is located in a directory different from the class file, we would use `getCodeBase` instead of `getDocumentBase` as follows:

```
import java.applet.*;
import java.io.*;
import java.net.*;

public class App2 extends Applet
{   public void init()
    {   try
        {   URL url = new URL(getCodeBase(), "sample.dat");
            InputStream in=url.openStream();
            /* do something with the input stream */
            in.close();
        }
        catch(MalformedURLException murle)
        {   System.err.println(murle); }
        catch(IOException ioe)
        {   System.err.println(ioe); }
    }
}
```

Now the applet will work as long as the HTML document it is on the same web server. In particular, the applet would work when loaded from your hard disk as long as the data file is available in the same directory as the applet code. ■

In short, using `getCodeBase()` or `getDocumentBase()` together with a `URL` object lets you construct applets that can read data either locally or from a web server, as long as both the applet and the data are located on the same machine.

Example 9.5.8:

In example 5.2.14 we created a `TickerTape` canvas that continuously moves a string of text from the right side of the screen to the left in a "ticker tape" fashion. Redo that example but this time the text displayed should be retrieved from a data file in the same directory as the applet. Also create a simple program to create the data file.

The second part of the example is easy: we create a program that writes the string given on the command line to a data file. We will use a `DataOutputStream` for this since the `URL` method `openStream` returns an `InputStream`, i.e. a byte-level input stream³²:

```
import java.io.*;

public class TickerData
{
    public static void main(String args[])
    {
        try
        {
            DataOutputStream out = new DataOutputStream(
                new FileOutputStream("ticker.dat"));
            out.writeUTF(args[0]);
            out.close();
            System.out.println("Wrote: '" + args[0] + "' to 'ticker.dat'");
        }
        catch(IOException ioe)
        {
            System.out.println("Error: " + ioe);
        }
    }
}
```

Instead of rewriting the `Ticker Canvas` example, here is a simplified version of that class, extending `Applet` for simplicity:

```
import java.awt.*;
import java.applet.*;

public class TickerTape extends Applet implements Runnable
{
    private int xPos, height, width;
    private Thread thread = null;
    private String text = "This is a Ticker Tape applet";
    private Font font = new Font("Monospaced", Font.BOLD, 12);
    public void init()
    {
        width = getFontMetrics(font).stringWidth(text);
        height = getFontMetrics(font).getHeight();
        xPos = getSize().width;
    }
    public void start()
    {
        if (thread == null)
        {
            thread = new Thread(this);
            thread.start();
        }
    }
    public void stop()
    {
        thread = null;
    }
    public void run()
    {
        Thread currentThread = Thread.currentThread();
        while (thread == currentThread)
        {
            if (xPos < -width)
                xPos = getSize().width;
            else
                xPos -= 2;
            repaint();
            try
            {
                thread.sleep(50);
            }
            catch (InterruptedException ie)
            {
                System.out.println(ie);
            }
        }
    }
}
```

³² If we could attach a character-based stream to an applet we could use a simple text editor to create the appropriate data file. In definition 9.5.9 we will see how to use a "bridge" class to connect an `InputStream` to a character-based file.

```

    }
}
public void paint(Graphics g)
{   g.setFont(font);
    g.drawString(text, xPos, height);
}
}

```

Note that our applet follows the recommendation for using the `Runnable` interface outline in definition 5.2.11, but it uses a hard-coded version of the string. To modify it so that the applet can read the text from the data file "ticker.dat", we will create a method `readText` as follows:

- we instantiate a new `URL`, using `getCodeBase` as the base address and the string "ticker.dat" as file name
- we use the method `openStream` on the `URL` object to get access to an input stream to the data file, and use that stream as input to a new `DataInputStream`
- we can then use the `readUTF` method of the `DataInputStream` to read the line from the data file

```

public void readText()
{   try
    {   URL url = new URL(getCodeBase(), "ticker.dat");
        DataInputStream in = new DataInputStream(url.openStream());
        text = in.readUTF();
        in.close();
    }
    catch(MalformedURLException murle)
    {   System.out.println(murle); }
    catch(IOException ioe)
    {   System.out.println(ioe); }
}

```

The last step is to add this method to the `TickerTape` applet and call it in the first line of the `init` method. Here is the `TickerTape` class again, with the new code in bold and italics:

```

import java.io.*;
import java.net.*;
import java.awt.*;
import java.applet.*;

public class TickerTape extends Applet implements Runnable
{   /* fields as before */
    public void init()
    {   readText();
        /* rest as before */
    }
    public void readText()           ( /* as defined above */ )
    public void start()             ( /* as defined before */ )
    public void stop()              ( /* as defined before */ )
    public void run()                ( /* as defined before */ )
    public void paint(Graphics g)    ( /* as defined before */ )
}

```

The text for this applet can now be modified by using the standalone program `TickerData` without recompiling the original applet.

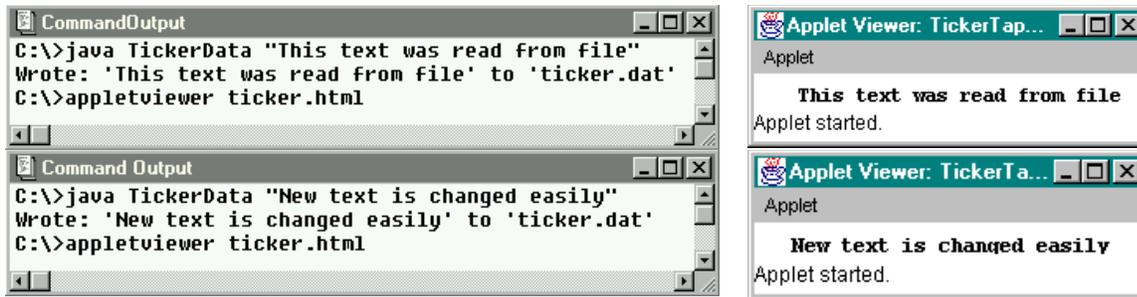


Figure 9.5.7: TickerTape reading data from file created by associated program

Bridges between Character and Byte Streams

One minor complication remains before we can "universally" use the `URL` class to open input streams: the `openStream` method returns an `InputStream` but if we want applets to read character data we need a `Reader` instead of an `InputStream`. But Java does, of course, provide classes that can act as a translator between byte streams and character streams.

Definition 9.5.9: The `InputStreamReader` Class

An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and translates them into characters. It uses the platform's default encoding, but another encoding may be specified. The Java API defines `InputStreamReader` as follows:

```
public class InputStreamReader extends Reader
{ // selected constructor
    public InputStreamReader(InputStream in)
}
```

As with a `FileReader`, an `InputStreamReader` can be wrapped inside a `BufferedReader` for better efficiency.

Since applets read their data through the web it usually pays off using a `BufferedReader`. After all, even the fastest connection to the Internet is orders of magnitude slower than a direct connection to a computer's hard disk, so the speed advantage that a buffered stream offers is good and welcome news.

Example 9.5.10:

In example 9.5.1 we created the `ReadAppletTest` to illustrate the restrictions that applets need to live with. Now change that applet so that it can correctly read a text string from a data file and display it in a label. Make sure to test the applet locally as well as through the web. The data file should be located in the same directory as the applet.

Recall the basic applet from example 9.5.1 (it is short, so we can repeat it here):

```
import java.applet.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.io.*;

public class ReadAppletTest extends Applet implements ActionListener
{   private Button read = new Button("Read");
    private Label display = new Label("Java by Definition");
    public void init()
    {   setLayout(new FlowLayout());
        add(read); read.addActionListener(this);
        add(display);
    }
    public void readFile()
    { /* old code using a FileReader giving a security exception */ }
    public void actionPerformed(ActionEvent ae)
    {   if (ae.getSource() == read)
        {   readFile();
        }
    }
}
```

Now we will use a URL object, an `InputStreamReader`, and a `BufferedReader` to read the data file in the `readFile` method (make sure to import all classes from the `java.net` package):

```
public void readFile()
{   try
    {   URL url = new URL(getCodeBase(), "st.dat");
        InputStreamReader ins = new InputStreamReader(url.openStream());
        BufferedReader in = new BufferedReader(ins);
        display.setText(in.readLine());
        in.close();
    }
    catch (MalformedURLException murle)
    {   System.err.println(murle); }
    catch (IOException ioe)
    {   System.err.println(ioe); }
}
```

This new applet will be able to read the string from the data file `st.dat` as long as the data file is located in the same directory as the applet itself. The data file can be created by any plain text editor. ■

Now that we know everything about sequential streams, we can also explain a class we introduced early in chapter two: the `Console` class:

8.6. System I/O Streams

In section 2.4, example 5, we introduced the `Console` class to obtain input from the keyboard. At that time we did not know anything about Java's exception handling mechanism, so we needed a class that would hide the fact that the Java standard input stream `System.in` contains only methods that throw exceptions. While the `Console` class takes a rather drastic approach to hide these exception - the program quits if an error occurs - it did fit the bill to make keyboard input simple for standalone non-GUI based programs.

We since learned how to deal with exception and now that we also understand the various stream types in the `java.io` package we are prepared to understand exactly how the `Console` class works.

First, let's look at the three standard system streams available whenever a Java program or applet starts:

Definition 9.7.1: The `System.in`, `System.out`, and `System.err` Streams

The `System` class contains three static fields representing streams that are available to every class. The streams are always open and ready to deliver or receive data:

- `System.out`: This stream is of type `PrintStream` and thus contains all methods of that stream (such as `print` and `println`). It delivers its output to the "standard output device", which is typically the DOS box or the Java console of a Web browser.
- `System.in`: This stream is of type `InputStream` and hence has limited functionality. It receives its input from the "standard input device", which is typically the keyboard (the DOS window usually must be the active window).
- `System.err`: This stream is of type `PrintStream` (like `System.out`) and is intended to deliver error messages to the "standard error device" even if the `System.out` stream has been redirected to another destination.

Three static methods from the `System` class are of interest with these streams:

- `void setIn(InputStream in)` *reassigns `System.in` to input stream `in`*
- `void setOut(PrintStream out)` *reassigns `System.out` to output stream `out`*
- `void setErr(PrintStream out)` *reassigns `System.err` to the output stream `out`*

This demystifies, hopefully, the standard input and output streams. In fact, since `System.out` and `System.in` are regular streams we can use standard stream-handling mechanisms to redirect or enhance the standard I/O streams.

Example 9.7.2:

Create a simple Java standalone program that uses the `Console` class from section 2.4, example 5, to obtain strings from a user. Depending on the command line parameter the program should write all strings to either a text file or to the screen until the user enters the single word "quit". If a file name is given as command line parameter, the output should be redirected to that file, otherwise the output should go to the screen.

We will later explain in detail how the `Console` class works. For now, we will simply use it, as usual. The new feature in this example is that if the command line input represents a valid file name, we will use the `System.setOut` method to redirect the standard output to that file. Otherwise we will leave it unchanged so that output by default will go to the screen. Important messages will be directed to `System.err` so that they will be visible even if standard output is redirected to a file.

```
import java.io.*;

public class Echo
{ public static void main(String args[])
  { PrintStream out = null;
    if (args.length == 1)
    { String fileName = args[0];
      try
      { out = new PrintStream(new FileOutputStream(fileName));
        System.out.println("Output redirected to " + fileName);
      }
    }
  }
}
```

```

        System.setOut(out);
    }
    catch(IOException ioe)
    { System.err.println(ioe); }
}
else
    System.out.println("Output not redirected to a file");
System.err.println("Enter the single word 'quit' to quit");
String s = new String();
while (!s.equals("quit"))
{ s = Console.readString();
  System.out.println(s);
}
if (out != null)
    out.close();
}
}

```

Note that the message "Enter the single word 'quit' to quit" is printed to `System.err` instead of `System.out` because `System.out` could already be redirected to the file, in which case the user would never see that message. ■

The ease of redirecting `System.out` or `System.err` can be quite useful for programs that need to print out status information or debugging strings. While developing such a program, the output could go to the screen via `System.out.println`, but the output can also be redirected to a file for later review by simply redirecting the `System.out` stream to a file.

As our last example we can now explain exactly how the `Console` class introduced in chapter two really works - and in fact, with all our knowledge at this point the class is completely easy to understand.

Example 9.7.3:

In section 2.4, example 5 we introduced the `Console` class to simplify getting input from the user in non-GUI programs. The class is defined below. Explain why and how it works.

Here is the definition of `Console` class again, without the javadoc comments that were part of the original class.

```

import java.io.*;

public class Console
{ public static double readDouble()
  { String tmpX = readString().trim();
    try
    { return (new Double(tmpX)).doubleValue(); }
    catch(NumberFormatException ne)
    { System.err.println("Console.readDouble: Not a double...");
      System.exit(-1);
      return 0.0;
    }
  }
}

public static int readInt()
{ String tmpX = readString().trim();
  try
  { return Integer.parseInt(tmpX); }
  catch(NumberFormatException ne)

```

```
        { System.err.println("Console.readInt: Not an integer...");
          System.exit(-1);
          return -1;
        }
    }
    public static String readString()
    { String string = new String();
      BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));
      try
      { string = in.readLine(); }
      catch(IOException e)
      { System.out.println("Console.readString: Unknown error...");
        System.exit(-1);
      }
      return string;
    }
}
```

The class has three methods: `readInt`, `readDouble`, and `readString`. The `readInt` and `readDouble` methods use `readString` to get input from the user, and try to convert it to a double or int, respectively. If an exception occurs during conversion, they halt without further ado any program using these methods.

The interesting method is `readString`. Since the `System.in` stream is an `InputStream`, it does not have any convenient methods to read formatted input, such as lines of text. In addition, all of its operations throw exceptions, which we would like to eliminate to keep simple programs as simple as possible. The `BufferedReader` stream, on the other hand, would be perfect for reading strings because of its `readLine` method. Therefore, we first use the `InputStreamReader` class to convert the byte-level `InputStream` `System.in` to a character-based `Reader` stream and then wrap that stream into a `BufferedReader`. That way we have a stream that is on one side attached to the standard input device (the keyboard), and internally converts the input bytes to format them as "lines" via the `readLine` method.

In an error occurs in these three methods, either an `IOException` or a `NumberFormatException`, we would *like* to create some code to handle it. But since our class can be used in many different situations, no single code could provide a meaningful solution. Thus we have not alternative but to stop the entire program in case an error happens. ■

If any program uses the methods provided by the `Console` class, the entire program will quit immediately as soon as an input error occurs. That approach is most definitely not going to allow for the creation of "robust" or "user-friendly" programs. But it does allow us to quickly get input from the user without the overhead of a GUI program and without having to worry about exceptions. The class, naturally, would not be used in "commercial" applications.

8.7. RandomAccessFile

There is one type of stream we did not explain in this chapter, and in fact will not explain at all: a `RandomAccessFile`. Instances of this class can support both read and write operations simultaneously. A random access file behaves much like a large array of bytes that happens to be stored in the file system. There is a kind of cursor, or index, to the implied array, called the file

pointer and input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If a random access file is created in read/write mode, then output operations are also available. Output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the `getFilePointer` method and set by the `seek` method.

It is generally true of all the reading routines in this class that if end-of-file is reached before the desired number of bytes has been read, an `EOFException` (which is a subclass of `IOException`) is thrown. If any byte cannot be read for any reason other than end-of-file, an `IOException` other than `EOFException` is thrown. In particular, an `IOException` may be thrown if the stream has been closed prior to reading or writing bytes. For further details on `RandomAccessFile`, please refer to the Java API.

Case Study: The Online Ordering Example

Now we have almost all ingredients to illustrate how an online ordering system could work. What will be missing is the final step of actually submitting the order, but we will complete the example in the next chapter.

Example 9.6.1:

Create an applet that will allow the user to pick items from an online catalog and order them online. The items in the catalog should include a name, an item ID, a description, an image, and a price. The user should be able to view the catalog and decide with items he or she wants. The user should be able to add items to an order, then submit the order by providing the necessary personal information. The entire applet should look reasonably professional. The data for the catalog should be stored in a data file read by the applet.

Note: the applet, at this time, does not actually have to submit the order.

Our idea for this online ordering system is as follows. First, we create a data file representing a catalog with items from which a potential customer can choose. We must use a stand-alone program for this task because the security restrictions for applets will not allow us to write data to disk. Then we create an applet that reads this catalog data file, presents the items in an appealing fashion and lets the user pick the items they want to order. Once an order is complete the customer can submit it via a dialog box. As mentioned, we will not be able to complete the actual submission of the order, but we will, for now, pretend that we could.

The first class to create is the one to represent an item in a catalog. That class is straightforward but for simplicity we will declare some fields of the class as protected so that we can access them directly without the help of standard `set` and `get` methods.

```

CatalogItem implements Serializable
private String ID;
private String shortName;
private String longName;
protected String imageName;
protected double price

CatalogItem(String _ID)
void setName(String _short, String _long)
String getName()
String toString()

```

Figure 9.6.1: Representation of CatalogItem Class

```

import java.io.*;

public class CatalogItem implements Serializable
{
    private String ID;
    private String shortName;
    private String longName;
    protected String imageName;
    protected double price;
    public CatalogItem(String _ID)
    {
        ID = _ID;
        shortName = longName = imageName = "";
        price = 0.0;
    }
    public void setName(String _shortName, String _longName)
    {
        shortName = _shortName;
        longName = _longName;
    }
    public String getName()
    {
        return ID + ": " + shortName;
    }
    public String toString()
    {
        return "ITEM " + getName() + "\n" + longName + "\n" +
            "Price: $" + price + "\n";
    }
}

```

Our `CatalogItem` implements `Serializable` so that it can be saved and retrieved via an object data stream. Next, we create a very simple standalone program to create a sample catalog. That program will not use a graphical user interface, but in the exercises you will be asked to create an improved GUI version of that program.

We will add six items to our catalog and we will assume that we have six images available in GIF format. The images are assumed to be located in a directory named `Images` located in the same directory where the class and data files are located.³³

```

import java.util.*;
import java.io.*;

public class CatalogAdmin
{
    public static void main(String args[])
    {
        LinkedList list = new LinkedList();
        try
        {
            FileOutputStream f_out = new FileOutputStream("catalog.dat");
            BufferedOutputStream b_out = new BufferedOutputStream(f_out);
            ObjectOutputStream out = new ObjectOutputStream(b_out);
            CatalogItem ci = new CatalogItem("001");

```

³³ The images are actually taken from one of the JDK 1.2 demo programs and can be found in the directory `jdk12\demo\jfc\SwingSet\images\ImageClub\food`

```

        ci.setName("Pepsi","Large Pepsi (32 Ounces) with colorful cup");
        ci.price = 2.99;    ci.imageName = "Images/softdrink.gif";
        list.add(ci);
        ci = new CatalogItem("002");
        ci.setName("French Fries","Large order of French Fries");
        ci.price = 1.59;    ci.imageName = "Images/fries.gif";
        list.add(ci);
        ci = new CatalogItem("003");
        ci.setName("Burger","1/4-pound Burger with Lectuce, Tomatos");
        ci.price = 1.99;    ci.imageName = "Images/burger.gif";
        list.add(ci);
        ci = new CatalogItem("004");
        ci.setName("Hot Dog","Giant all meat Hot Dog with condiments");
        ci.price = 1.29;    ci.imageName ="Images/hotdog.gif";
        list.add(ci);
        ci = new CatalogItem("005");
        ci.setName("Ice Cream","Large serving of Vanilla Ice Cream");
        ci.price = 2.99;    ci.imageName = "Images/icecream.gif";
        list.add(ci);
        ci = new CatalogItem("006");
        ci.setName("Pizza","Pizza Slice (tomatoes, cheese, peperoni)");
        ci.price = 1.79;    ci.imageName = "Images/pizza.gif";
        list.add(ci);
        out.writeObject(list);
        out.close();
    }
    catch(IOException ioe)
    { System.out.println(ioe); }
}
}

```

Before we continue, you should compile and execute `CatalogAdmin` to create the data file named `catalog.dat`. The images must be copied manually into the directory `Images`, located in the same directory where the class and data files are.

Our applet is supposed to let the user pick items from the catalog to put together an order. Therefore, before worrying about the actual applet, we will first create another data storage class to hold an "order". That class will use a `LinkedList` to store the items the customer wants to order, together with billing and mailing information from the customer to complete the sale.

```

CatalogOrder implements Serializable
private String name;
private String credit;
private String address;
private LinkedList list;
private double totalCost;

void setInfo(String _name, String _credit,
             String _address)
void add(CatalogItem item)
String toString()

```

Figure 9.6.2: Representation of CatalogOrder Class

Again, this class is straightforward and will also implement the `Serializable` interface so that we can later submit the order via an object stream:

```

import java.util.*;
import java.io.*;

public class CatalogOrder implements Serializable

```

```

{ private String name = null;
  private String credit = null;
  private String address = null;
  private LinkedList list = new LinkedList();
  private double totalCost = 0.0;
  public void setInfo(String _name, String _credit, String _address)
  { name = _name;
    credit = _credit;
    address = _address;
  }
  public boolean isComplete()
  { return (!(name.equals("")) &&
            (!credit.equals("")) &&
            (!address.equals("")));
  }
  public void add(CatalogItem item)
  { list.add(item);
    totalCost += item.price;
  }
  public String toString()
  { String s = "Total items: " + list.size() + "\n";
    s += "Total cost: $" + totalCost + "\n\n";
    for (Iterator iterator = list.iterator(); iterator.hasNext(); )
      s += "Item " + ((CatalogItem)iterator.next()).getName() + "\n";
    return s;
  }
}

```

With these preliminary data storage classes out of the way, the real work of creating an appealing applet starts. The idea is simple: the applet will read the catalog, add the items to a list, allow the user to see the full description of an item together with its image, and let the user pick items to put together an order. Since our applet is supposed to display an image for each item in the catalog, we need a class that can load and display images for us so that we can position the image with a layout manager. Therefore, we next create the `CatalogImage` class³⁴ as follows:

CatalogImage	extends Canvas
<pre> private String image; void setImage(URL base, String _image) void Dimension getPreferredSize() void paint(Graphics g) </pre>	

Figure 9.6.3: Representation of `CatalogImage` Class

```

import java.net.*;
import java.awt.*;

public class CatalogImage extends Canvas
{ private Image image = null;
  public CatalogImage()
  { super(); }
  public void setImage(URL base, String name)
  { try
    { URL url = new URL(base, name);
      image = Toolkit.getDefaultToolkit().getImage(url);
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(image, 0);
      tracker.waitForAll();
    }
  }
}

```

³⁴ If we used Swing instead of AWT we could simply use a `JLabel`. It already has all the functionality we need (and more).

```

        catch(Exception e)
        { System.err.println("Image not available or not loaded" + e);
          image = null;
        }
        repaint();
    }
    public Dimension getPreferredSize()
    { return new Dimension(100, 100); }
    public void paint(Graphics g)
    { g.drawRect(0,0,getSize().width-1, getSize().height-1);
      if ((image != null) && (image.getWidth(this) > 0))
      { int x = (getSize().width - image.getWidth(this)) / 2;
        int y = (getSize().height - image.getHeight(this)) / 2;
        g.drawImage(image, x, y, this);
      }
      else
      { g.setColor(Color.red);
        g.drawLine(0, 0, getSize().width, getSize().height);
        g.drawLine(getSize().width, 0, 0, getSize().height);
      }
    }
}

```

The `setImage` method takes as input a URL representing the base location of our images and the name of the image to load. It then uses the `getImage` method from the `Toolkit` class to load the image. We use a `MediaTracker` to ensure that the image is completely loaded before proceeding. There are two exceptions that could be thrown in this class: a `MalformedURLException` (by the URL constructor) and an `InterruptedException` (by the `MediaTracker`), and we will capture both of them in one `catch` clause for simplicity. The `getPreferredSize` will inform a layout manager that these images would like a canvas of size 100 by 100 pixels so the actual images should be no larger than this size. Finally, the `paint` method will either draw the image centered within the canvas if possible, or it will draw a large, red "x" inside the canvas if the image can not be found or an error occurs during the loading process.

In effect, this class will handle all necessary details to load and display the image inside a canvas. All we have to do is create an instance of this class and pass the image name into the `getImage` method. You may want to refer to chapter six for details on loading and displaying images.

Now we are almost ready to create our final applet. However, it is supposed to look (semi) professional, so we need a way to improve the basic look of our applet. We will develop a class `PanelBox` that will automatically put a 3D-like border around a component together with a title. Such a class is simple to create and use, and will with very little effort greatly improve the looks of our applet.³⁵

```

PanelBox extends Panel
private Component component;
private Color bgColor;
private String label;

PanelBox(Component _component, String _label)
PanelBox(Component _component,
          String _label, Color _bgColor)
Insets getInsets()
void paint(Graphics g)

```

Figure 9.6.4: Representation of `PanelBox` Class

³⁵ Since Swing components can automatically have a border and a title, it would again be easier to create our program using Swing instead of AWT.

In fact, the `PanelBox` class will be reusable and can be used in other applets or frames as well:

```
import java.awt.*;

public class PanelBox extends Panel
{   private int h = 0, w = 0, d = 0;
    private Component component = null;
    private Color bgColor = Color.white;
    private String label = "Panel";
    private Font font = new Font("Helvetica", Font.BOLD, 12);
    public PanelBox(Component _component, String _label)
    {   super();
        component = _component; label = _label;
        setup();
    }
    public PanelBox(Component _component, String _label, Color _bgColor)
    {   super();
        component = _component; label = _label; bgColor = _bgColor;
        setup();
    }
    public Insets getInsets()
    {   return new Insets(h+1, h+1, h+1, h+1); }
    public void paint(Graphics g)
    {   g.setColor(Color.gray);
        g.drawRect(d, d, getSize().width-2*d, getSize().height-2*d);
        g.setColor(Color.black);
        g.drawLine(getSize().width-d+1, d,
                   getSize().width-d+1, getSize().height-d+1);
        g.drawLine(d, getSize().height-d+1,
                   getSize().width-d+1, getSize().height-d+1);
        g.setFont(font);
        g.setColor(bgColor);
        g.fillRect((int)(1.5 * h)-5,0, w+10, h+2);
        g.setColor(Color.black);
        g.drawString(label, (int)(1.5 * h), h-2);
    }
    private void setup()
    {   super.setBackground(bgColor);
        setBackground(bgColor);
        setLayout(new BorderLayout());
        add("Center", component);
        w = getFontMetrics(font).stringWidth(label);
        h = getFontMetrics(font).getMaxAscent() + 1;
        d = h/2;
    }
}
```

The `setup` method adds the input `component` to the center of the panel and determines the height and width of the title string of our box in the specified font. The class also contains the method `getInsets` that the component inherits and that we override here. That method determines how much padding the `component` should receive when it is laid out by a layout manager. In our case we give the panel as much padding as the height of the text in the specified font. That ensures that the component will actually be inside the border that is drawn by the `paint` method. The `paint` method does most of the work. It first draws a gray rectangle around the component, then adds black lines around the right and bottom side to give the rectangle a 3D look. Next, we draw a small rectangle in the background color to erase part of the rectangle just drawn. The dimensions of that rectangle are chosen so that it can contain the title of our box, which is drawn next in the specified font.

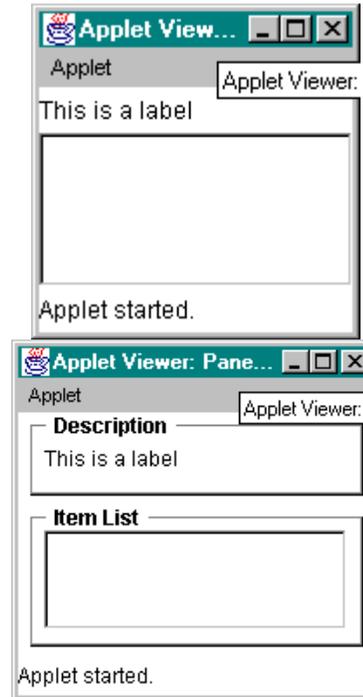
This class is very simple to use, yet adds a nice look to any component. Here is a quick example before we continue with our main task of creating the catalog applet:

Without PanelBox:

```
public class PanelBoxTest extends Applet
{   Label label=new Label("This is a Label");
    List list  = new List(3);
    public void init()
    {   lable =
        setLayout(new BorderLayout());
        add("Center", list);
        add("North", label);
    }
}
```

With PanelBox:

```
public class PanelBoxTest extends Applet
{   Label label=new Label("This is a Label");
    List list  = new List(3);
    public void init()
    {   setLayout(new BorderLayout());
        add("Center",new PanelBox(list,"Item List"));
        add("North",new PanelBox(label,"Description"));
    }
}
```



Now we are finally ready for our main applet. We will use a `LinkedList` to store the items from the catalog read from disk and a `List` (from `java.awt`) to display a list of items.³⁶ We will also add two text areas to our applet, one to display the full description of an item selected from the list and another to display the items currently ordered. Finally, we will use the `CatalogImage` class to display the pictures associated with a particular item, and wrap all components inside `PanelBoxes` with appropriate labels. Here is the outline of our class, including the layout of the various components:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.io.*;
import java.net.*;

public class Catalog extends Applet implements ActionListener
{   private Button orderButton  = new Button("Order Item");
    private Button submitButton = new Button("Submit Order");
    private Button clearButton  = new Button("Clear Order");
    private java.awt.List catalog = new java.awt.List(5);
    private TextArea description = new TextArea(5,40);
    private TextArea orders     = new TextArea(5,30);
    private LinkedList data     = null;
    private CatalogImage theImage = new CatalogImage();
    private CatalogOrder theBill  = new CatalogOrder();
    public void init()
    {   Color color = new Color(255,255,200);
        Panel items = new Panel();
        items.setLayout(new BorderLayout());
```

³⁶ If we used a `JList` from `javax.swing` we could store `CatalogItem` objects directly, using the model attached to the `JList`.

```

        items.add("West", new PanelBox(catalog, "Catalog", color));
        items.add("Center", new PanelBox(orders, "Your Order", color));
        Panel buttons = new Panel();
        buttons.setLayout(new FlowLayout());
        buttons.add(orderButton); orderButton.addActionListener(this);
        buttons.add(submitButton); submitButton.addActionListener(this);
        buttons.add(clearButton); clearButton.addActionListener(this);
        Panel panel = new Panel();
        panel.setLayout(new BorderLayout());
        panel.add("West", theImage);
        panel.add("Center", description);
        setLayout(new BorderLayout());
        add("North", items);
        add("Center", new PanelBox(panel, "Description", color));
        add("South", new PanelBox(buttons, "Click a button", color));
    }
    public void readCatalog()
    { /* reads the catalog from disk into data list and populates the
        catalog java.awt.list */ }
    public void showDescription(int i)
    { /* shows full description and image for a selected item */ }
    public void orderItem(int i)
    { /* adds the selected item to theBill and shows order so far */ }
    public void clearOrders()
    { /* resets theBill and clears the order */ }
    public void actionPerformed(ActionEvent ae)
    { /* react to button clicks */ }
}

```

The applet will compile (but it will of course do nothing so far), producing look shown in figure 9.6.5:

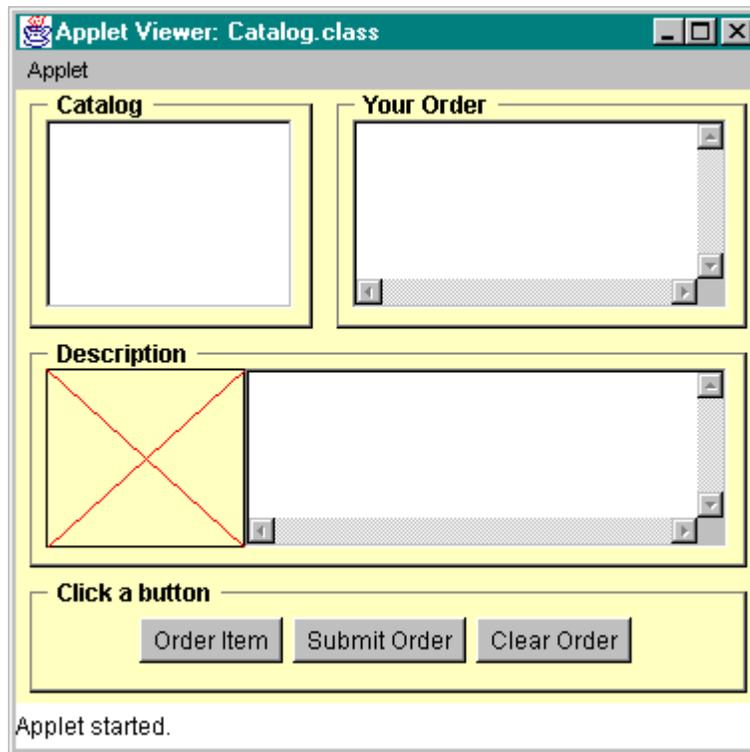


Figure 9.6.5: Layout for Catalog applet

The `CatalogImage` produces a large, red "X" since it does not have an image to display.

Next, we need to figure out how exactly the applet should work. The most complicated method is the `readCatalog` method: it opens an object stream via a `FileInputStream` to the data file `catalog.dat`, reads the entire catalog into the data list, and populates the `java.awt.List` `catalog`. It also calls `clearOrders` to initialize an empty `theBill` object:

```
public void readCatalog()
{ try
  { URL url = new URL(getCodeBase(), "catalog.dat");
    BufferedInputStream bin= new BufferedInputStream(url.openStream());
    ObjectInputStream in = new ObjectInputStream(bin);
    data = (LinkedList)in.readObject();
    for (Iterator iterator=data.iterator(); iterator.hasNext(); )
      catalog.add(((CatalogItem)iterator.next()).getName());
    in.close();
    clearOrders();
  }
  catch(Exception e)
  { catalog.add("not available");
    description.setText("Catalog not available, try again later");
    System.err.println(e);
  }
}
```

The remaining methods are all simple: `showDescription` takes as input an integer representing which item in the list was selected. It retrieves the corresponding object from the data list, sets the text in `description` by using the `toString` method of the `CatalogItem`, and sets the image of the `CatalogImage` to the name specified in the catalog item:

```
public void showDescription(int i)
{ CatalogItem item = (CatalogItem)data.get(i);
  description.setText(item.toString());
  theImage.setImage(getCodeBase(), item.imageName);
}
```

The `orderItem` method takes again an integer input representing which item in the list was selected. It then adds that item to `theBill`, sets the text of orders by using the `toString` method of `theBill`, and enables the `submitButton` so that an order could now be submitted by the user:

```
public void orderItem(int i)
{ theBill.add((CatalogItem)data.get(i));
  orders.setText(theBill.toString());
  submitButton.setEnabled(true);
}
```

The `clearOrders` method instantiates a new `CatalogOrder` and assigns it to `theBill`, sets the text in `orders` to "no order", and disables the `submitButton` so that an empty order can not be submitted.

```
public void clearOrders()
{ theBill = new CatalogOrder();
  orders.setText("No items selected for purchasing");
  submitButton.setEnabled(false);
}
```

Of course, still nothing will happen until we implement the `actionPerformed` method. Actually, we can do slightly better: we want some action to happen when the user clicks on a button, but also if the user selects a new item from the catalog list. In fact, if the user selects a new item from the list by clicking once (or using the arrow keys), we want to update the description of the newly selected

catalog item. If the user double-clicks on an item in the list, we want that to mean that the user wants to order that item.

To achieve the first goal (react to a new selection in the list), we will change the definition of our class so that it also implements `ItemListener`. Of course that means that we need to implement the `itemStateChanged` method as well. Then we can add an `ItemListener` to the `catalog` list to react to a new selection in the list, as well as an `ActionListener` to react to a double-click on a list item. Adding the `ActionListener` to the `catalog` list will, of course, achieve the second goal. Finally, we need to make sure our catalog data file is loaded at the time the applet is constructed, so we add a call to `readCatalog` to the end of the `init` method so that the catalog is loaded when the applet is initialized. Our new class will look as follows:

```
Catalog
extends Applet
implements ActionListener, ItemListener

Buttons order, submit, clear;
TextAreas description, orders
List catalog;
LinkedList data;
CatalogImage theImage;
CatalogOrder theBill;
void init()
void readCatalog()
void showDescription(int i)
void orderItem(int i)
void clearOrders()
void actionPerformed(ActionEvent ae)
void itemStateChanged(ItemEvent ie)
```

Figure 9.6.6: Representation of Catalog Class

```
/* import packages as before */

public class Catalog extends Applet
    implements ActionListener, ItemListener
{ /* all fields as before */
    public void init()
    { /* everything as before, but with three lines added at the end */
        catalog.addActionListener(this);
        catalog.addItemListener(this);
        readCatalog();
    }
    /* all methods as before, and in addition the following methods */
    public void actionPerformed(ActionEvent ae)
    { if ((ae.getSource() == orderButton) || (ae.getSource() == catalog))
      { if (catalog.getSelectedIndex() >= 0)
        orderItem(catalog.getSelectedIndex());
      }
      else if (ae.getSource() == clearButton)
        clearOrders();
    }
    public void itemStateChanged(ItemEvent ie)
    { showDescription(catalog.getSelectedIndex()); }
}
```

Our final task to complete this example is to implement a way by which the order can be submitted. For a valid order, we need to ask the user for credit and personal information, and we should give him or her another chance to review their order before finally sending it.

The class perfect for this task would be the `Dialog` class. However, a `Dialog` can not be used by an `Applet`, because the `Dialog` constructor requires a reference to a `Frame` as input. Hence, we will use a

Frame instead of a `Dialog` to collect personal and credit information. Because frames are never modal (i.e. a user can open a new frame and continue to work in the original frame or applet) we need to make sure that a user can open up exactly one such order submission frame at a time. Therefore, the new frame will get a reference to its parent applet and rely on a `closeSubmission` method that we need to implement in our applet for closing the submission frame. We will also use the `PanelBox` class again to improve the look of the submission frame. The rest of the class is simple, particularly because we can not yet implement the code for actually submitting the order. Here is the class:

```
CatalogSubmission
extends Frame
implements ActionListener

Catalog owner;
TextFields name, credit;
TextAreas address, orders;
CatalogOrder theBill;
Buttons submit, cancel;
Label status

CatalogSubmission(Catalog _owner,
                  CatalogOrder _theBill)
void actionPerformed(ActionEvent ae)
```

Figure 9.6.7: Representation of `CatalogSubmission` Class

```
import java.awt.*;
import java.awt.event.*;

public class CatalogSubmission extends Frame implements ActionListener
{
    private Catalog owner = null;
    private TextField name = new TextField();
    private TextArea address = new TextArea(3,20);
    private TextField credit = new TextField();
    private TextArea orders = new TextArea(6,30);
    private CatalogOrder theBill = null;
    private Button submit = new Button("Submit Order");
    private Button cancel = new Button("Cancel Order");
    private Label status = new Label("Enter personal and mailing info");
    public CatalogSubmission(Catalog _owner, CatalogOrder _theBill)
    {
        super("Submit your Order");
        owner = _owner;
        theBill = _theBill;
        Color color = new Color(255,255,200);
        Panel buttons = new Panel(); buttons.setLayout(new FlowLayout());
        buttons.add(submit); submit.addActionListener(this);
        buttons.add(cancel); cancel.addActionListener(this);
        Panel cInfo = new Panel(); cInfo.setLayout(new GridLayout(2,2));
        cInfo.add(new Label("Full Name:"));
        cInfo.add(name);
        cInfo.add(new Label("Credit Card: "));
        cInfo.add(credit);
        Panel pInfo = new Panel(); pInfo.setLayout(new BorderLayout());
        pInfo.add("West",new PanelBox(cInfo,"Personal Info",color));
        pInfo.add("East", new PanelBox(address, "Mailing Address", color));
        Panel oInfo = new Panel(); oInfo.setLayout(new BorderLayout());
        oInfo.add("Center", new PanelBox(orders, "Your order: ", color));
        oInfo.add("South", new PanelBox(status, "Status:", color));
        setLayout(new BorderLayout());
        add("North", pInfo); add("Center", oInfo); add("South", buttons);
        orders.setText(theBill.toString()); orders.setEditable(false);
        validate(); pack(); setVisible(true);
        submit.setEnabled(true);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
```

```

        { owner.closeSubmission(); }
    });
}
public void actionPerformed(ActionEvent ae)
{
    if (ae.getSource() == cancel)
        owner.closeSubmission();
    else if (ae.getSource() == submit)
    {
        theBill.setInfo(name.getText(), credit.getText(),
            address.getText());
        if (theBill.isComplete())
            status.setText("NOT SUBMITTED: feature not implemented.");
        else
            status.setText("NOT SUBMITTED: incomplete information.");
    }
}
}
}

```

Note that we are using an anonymous inner class to activate the close box of our frame. To use this "dialog frame", we need to make some final modifications to our `Catalog` applet:

1. Add a field of type `CatalogSubmission` to the class `Catalog` as follows:

```
private CatalogSubmission submission = null;
```

2. Modify the `actionPerformed` method of `Catalog` as follows:

```
public void actionPerformed(ActionEvent ae)
{
    if ((ae.getSource() == orderButton) || (ae.getSource() == catalog))
    {
        if (catalog.getSelectedIndex() >= 0)
            orderItem(catalog.getSelectedIndex());
    }
    else if (ae.getSource() == clearButton)
        clearOrders();
    else if (ae.getSource() == submitButton)
    {
        if (submission == null)
            submission = new CatalogSubmission(this, theBill);
        else
            submission.toFront();
    }
}
}

```

3. Add a method `closeSubmission()` to `Catalog` as follows:

```
public void closeSubmission()
{
    submission.setVisible(false);
    submission.dispose();
    submission = null;
}

```

```

Catalog extends Applet
    implements ActionListener, ItemListener

    Buttons order, submit, clear;
    TextAreas description, orders
    List catalog;
    LinkedList data;
    CatalogImage theImage;
    CatalogOrder theBill
    CatalogSubmssion submission;

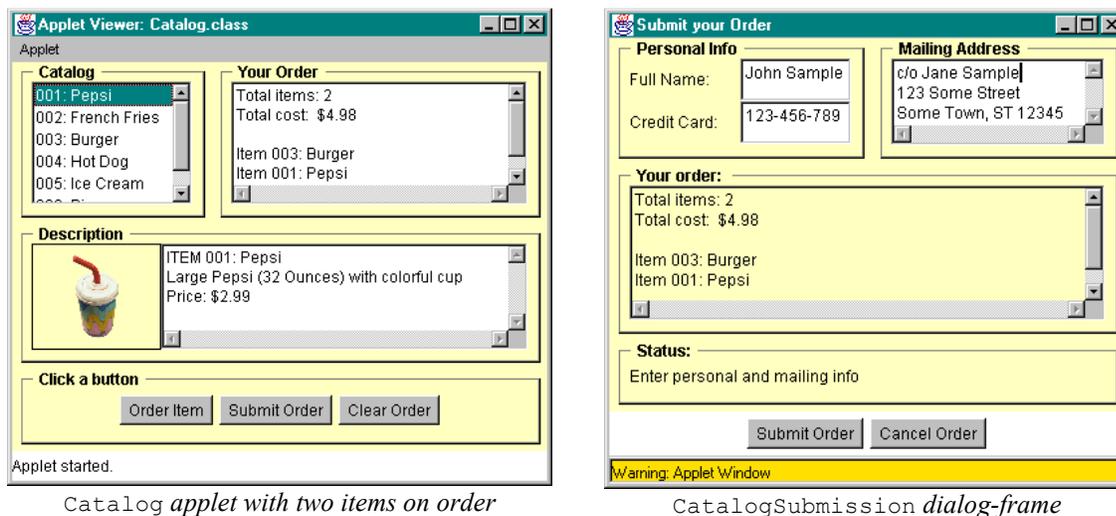
    void init()
    void readCatalog()
    void showDescription(int i)
    void orderItem(int i)
    void clearOrders()
    void closeSubmission()
    void actionPerformed(ActionEvent ae)
    void itemStateChanged(ItemEvent ie)

```

Figure 9.6.8: Representation of final `Catalog` class

These modifications mean that when the `submit` button is clicked, the applet first checks if a `CatalogSubmission` frame is already active. If so it will be brought to the foreground, otherwise a new instance is created. On the other hand, if the `cancel` or `submit` buttons are pressed, the `closeSubmission` method of the applet is called to close the submission frame and set `submission` to null so that another instance can be instantiated later.

All of these classes and modifications finally finish this example. Make sure that all class files are located in the same directory, and the images are located in a separated directory called `Images`, also in the same directory. Then compile and run the `CatalogAdmin` class first to create the data file before running the `Catalog` applet (via the appropriate `HTML` file and the appletviewer. Figure 9.6.9 shows some sample screen shots of what you might see:



Catalog applet with two items on order

CatalogSubmission dialog-frame

Figure 9.6.9: Catalog applet with submission dialog

Chapter Summary

